PATENT

Paper No.

Our File No.: AIS-P99-1

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Inventor	:	MARKS, Daniel L.
Serial No.	:	09/399,578
Filed	:	September 20, 1999
For	:	GROUP COMMUNICATIONS MULTIPLEXING SYSTEM
Group Art Unit	:	2452
Confirmation No.	:	2427
Examiner	:	WINDER, Patrice L.

MS: Fee Amendment
The Commissioner of Patents
P.O. Box 1450
Alexandria, VA 22313-1450

DECLARATION OF DR. CHANDRAJIT BAJAJ

S I R :

1. My name is Chandrajit Bajaj. I am a Computational Applied Mathematics Chair in Visualization, Professor of Computer Sciences, and Director of the Center for Computational Visualization at the Institute of Computational Engineering and Sciences, University of Texas at Austin, where I have been a faculty member since 1997. My resume is provided herewith.

2. I am a co-author of the article titled “SHASTRA - An Architecture for Development of Collaborative Applications” (‘Shastra’).

3. The other co-author is Vinod Anupam, who was my graduate student when I was a Professor of Computer of Science and the Director of the Center for Image Analysis and Visualization, at Purdue University.

4. I was the chairman of the Ph. D Dissertation Committee for Vinod Anupam, one of his main advisors, and a signer of his Dissertation.

5. Under my direct supervision, Vinod Anupam wrote the computer code which was the basis of his Dissertation, this code being provided herewith.

6. In view of my familiarity with the Shastra article, the Dissertation, and the code ('Shastra system'), I have been retained to provide information regarding these in this declaration, for which I am being compensated.

7. As evidenced by the code, there was no capability of "a database which serves as a repository of tokens for other programs to access, thereby affording information to otherwise independent participator computers."

8. This capability also is not disclosed in the Shastra article, the Dissertation, or any co-authored prior art articles about the Shastra system.

9. This capability would not have been contemplated for the reason that adding the capability would defeat the otherwise contemplated objective of collaborative multimedia, as set out in the title and otherwise discussed in the Dissertation.

10. I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true, and further that these statements were made with knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code, and that such willful false statement may jeopardize the validity of the application or any patent issued thereon.

Date: July 14, 2011

Chandrajit Bajaj, Ph.D.

Biographical Sketch: Chandrajit L. Bajaj

Dr. Bajaj is Computational Applied Mathematics Chair in Visualization, Professor of Computer Sciences, and Director of the Center for Computational Visualization at the Institute of Computational Engineering and Sciences, University of Texas at Austin.
(http://www.cs.utexas.edu/~bajaj and http://www.ices.utexas.edu/~bajaj)

Research Areas

Dr. Bajaj's research areas of interest include Image Processing, Computational Geometry, Geometric Modeling, Computer Graphics, Visualization, Computational Biology and Bioinformatics. He is currently involved in developing integrated approaches to computational modeling, mathematical analysis and interrogative visualization, especially for dynamic bio-medical structures and phenomena.

Education

- B.Tech. in Electrical Engineering Indian Institute of Technology, Delhi 1980
- M.S in Computer Science Cornell University 1983
- Ph.D. in Computer Science Cornell University 1984

Professional Experience

- Assistant Professor of Computer Science, Purdue University, 1984-89
- Associate Professor of Computer Science, Purdue University, 1989-93
- Visiting Associate Professor of Computer Science, Cornell University, 1990-91
- Professor of Computer Sciences, Purdue University, 1993-97
- Director of Image Analysis and Visualization Center, Purdue University, 1996-97
- Computational Applied Mathematics Chair in Visualization, University of Texas, 1997-
- Professor of Computer Sciences, University of Texas 1997-
- Director of Center for Computational Visualization, University of Texas, 1997-

Recent Selected Honors and Awards

- Best paper award at Computer Aided Design (CAD) 2006
- Panel Member of the National Academy of Sciences, Vietnam Education Foundation, 2006, 2007
- Member of the NSF-CISE Board of Visitors, 2004, ETH Zurich, CS Dept Evaluation Committee (2004), INRIA Evaluation Committee 2007
- King Abdullah University of Science and Technology Center Director Chair search committee, 2008
- Member of Consolider Scientific Committee of the Spanish Ministerio de Ciencia e Innovacion, 2008, 2009
- Member of the NIH-NCRR National Biomedical Computation Resource Advisory Committee, 2006 –
- Charter Member of Molecular Structure Function (MSFD) "Computational BioPhysics" Study Section, National Institute of Health, 2008 –
- Fellow of the Association for Computing Machinery (ACM), 2009 –
- Fellow of the American Association for the Advancement of Science (AAAS), 2008 –
- Moncrief Grand Challenge Faculty Award, 2009
- Fellow of the Association of Computing Machinery (ACM), 2009 -

Ten Significant Publications (out of over 200 publications in full CV)

1. "Volumetric Feature Extraction and Visualization of Tomographic Molecular Imaging", (with Z. Yu, M. Auer), Journal *of Structural Biology*, 144: 1-2, (2003), 132-143.

2. "Automatic Ultra-structure Segmentation of Reconstructed Cryo-EM Maps of Icosahedral Viruses", (with Z. Yu), *IEEE Transactions on Image Processing: Special Issue on Molecular and Cellular Bioimaging*, Sep;14, 9, (2005):1324-37
3. "Computational Approaches for Automatic Structural Analysis of Large Bio-molecular Complexes", (with Z. Yu), *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, (digital library) 22 June 2007, PMID: 18989044, PMC Journal in Process
4. "F2Dock: Fast Fourier Protein-Protein Docking", (with R. Chowdhury, V. Siddahanavalli), IEEE/ACM Transactions on Computational Biology and Bioinformatics, 2009, Accepted for publication, NIHMSID153460, PMC Journal in Process
5. "Fast Molecular Solvation Energetics and Force Computation", (with W. Zhao), SIAM Journal on Scientific Computing, Accepted for publication, NIHMSID153453, PMC Journal in Process
6. "Anisotropic Diffusion of Surfaces and Functions on Surfaces" (with G. Xu), *ACM Transactions on Graphics*, 22:1, (2003), 4 – 32.
7. "Dynamic Maintenance and Visualization of Molecular Surfaces", (with V. Pascucci, A. Shamir, R. Holt, A. Netravali), Fourth issue in the special series of Discrete Applied Mathematics on Computational Molecular Biology, 127, (2003), 24 –51.
8. "Application of New Multiresolution Methods for the Comparison of Biomolecular Electrostatic Properties in the Absence of Structural Similarity", (with N. Baker, B. Kwon, T. Dolinsky, J. Nielsen, X. Zhang), *Multiscale Modeling and Simulation,* (2006), 5 (4), 1196-1213, PMID: 18841247, PMCID: PMC2561295
9. "Geometric Modeling and Quantitative Visualization of Virus Ultrastructure", *Modeling Biology: Structures, Behaviors, Evolution*, ed. by L. da Fontoura Costa, M. Laublichler, MIT Press, 2007.
10. "The Capsid Proteins of a Large, Icosahedral dsDNA Virus", (with X. Yan, Z. Yu, P. Zhang, A. Batistti, P. Chipman, M. Bergoin, M. Rossman and T. Baker), *Journal of Molecular Biology*, Available online from doi:10.1016/j.jmb.2008.11.002, PMID: 19027752, PMC Journal in Process

Synergistic Activities

- Editor, ACM Transactions on Graphics, 1995 –
- Editor, International Journal of Computational Geometry and Applications, 1994 –
- Editor, ACM Computer Surveys, 2004 –
- Editor, Computational Vision and Biomechanics, 2006 –
- Editor, SIAM Journal of Imaging Sciences, 2007 –
- Chair, Intl. Symposium on Symbolic and Algebraic Computation (ISSAC), UK, 2000, and ACM Annual Symposium on Computational Geometry (Applied Track), 2002
- Keynote Addresses at Computer Graphics 2002, Volume Graphics 2003, EuroGraphics 2003, Cyberworlds 2005, Jacques Morgenstern Colloquium, INRIA-Sophia Antipolis 2006, Institute of Mathematics and Applications 2007, Computational Modeling of Objects Presented in Images , 2010

Expert Witness/Patent Infringement Cases in Recent Years

- Patent Case, American Video Graphics, L.P. versus Electron Arts Inc., et al, 2004-2005
- Patent Case, Landmark Graphics Corp. and Magic Earth Inc. versus Seismic Microtechnology, 2006
- Patent Case, Paradigm Geophysical Corp. versus Magic Earth, 2006
- Patent Case, Symbol Technologies versus Metrologic Instruments, 2006-2007

Industry Consultancy in Recent Years

- AT&T Research Labs, Murray Hill, NJ
- Earth Simulator Center, Yokohama, Japan
- eCalibre Technologies Inc., Austin, TX
- Lucent Technologies, Murray Hill, NJ
- Shinko Denki Corp., Ise, Japan
- Seimens Research Corp., Princeton, NJ
- Schlumberger Research Corp., Austin, TX

Order Number 9512924

Collaborative multimedia environments for problem-solving

Anupam, Vinod, Ph.D.

Purdue University, 1994

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

4/319
7-26-94

Graduate School Form 9
(Revised 8/89)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis Acceptance

This is to certify that the thesis prepared

By Vinod Anupam

Entitled
Collaborative Multimedia Environments For Problem Solving

Complies with University regulations and meets the standards of the Graduate School for originality and quality

For the degree of Doctor of Philosophy

Signed by the final examining committee:

_____, chair

Approved by:

John R. Rice / William J. Gorman 25 July 1994
Department Head Date

☐ is
This thesis ☒ is not to be regarded as confidential

Major Professor

COLLABORATIVE MULTIMEDIA ENVIRONMENTS FOR PROBLEM SOLVING

A Thesis
Submitted to the Faculty

of

Purdue University

by

Vinod Anupam

In Partial Fulfillment of the
Requirements for the Degree

of

Doctor of Philosophy

August 1994

To my parents..

ACKNOWLEDGMENTS

I would like to thank my advisor, Prof. Chandrajit Bajaj, for many things, culminating in this thesis : for giving me great latitude in this exploratory process; for his constant support, encouragement and guidance; for the many fruitful discussions we have had; and for his invaluable help both within and outside the world of academics.

To my many teachers through the years, especially at Sainik School Ghorakhal, at Kendriya Vidyalaya Jalahalli, at Birla Institute of Technology and Science (BITS) Pilani, and at Purdue University, I owe a heartfelt debt of gratitude. They led me along the path of knowledge, and taught me to learn and grow.

For all that I am and all that I have, I thank my parents. My father, Vinod Prakash Saxena, and my mother, Manjulika Saxena, inculcated in me a thirst for knowledge, stressed on the importance of a well rounded education, and encouraged me to strive to excel – to be all I could be. To my wife, my life, Mary – thanks for your patience and perseverance, and for all your help. I thank my twin brother Atulya and my younger brother Ashwini for all the learning we have done together.

Many thanks to all my collaborators and lab-mates through the times – Steve Klinkner, Dr. Tamal Dey, Dr. Insung Ihm, Dr. Andrew Royappa, Steve Cutchin, Jindon Chen, Susan Evans, Dan Schikore, Fausto Bernardini, Peinan Zhang, Dr. Xu Guoliang, Dr. Youming Lin, and Dr. Zhang Xuan. We have had many stimulating discussions on a multitude of topics, often not academic. Too numerous to thank are the many friends I made during the course of my stay at Purdue. They made sure there was a life besides academic pursuits. And many thanks to all my squash buddies for our frequent encounters on the courts of the Co-Rec.

My graduate committee, Prof. John Rice, Prof. Aditya Mathur and Prof. Vincent Russo, contributed valuable advice. Prof. Dan Marinescu and Prof. Prasun Dewan offered much input and constructive criticism.

I gratefully acknowledge the intellectual and material support provided by the fine environs and the great staff of the Computer Sciences Department of Purdue University. Rich Bingle, Doug Crabill, Adam Hammer and Dan Trinkle were always available to help get troublesome demons out of the system. Georgia Conarroe, Patti Minniear and Daloris Williamson provided much infrastructural support.

TABLE OF CONTENTS

Page

Page

Page

LIST OF FIGURES

ABSTRACT

Anupam, Vinod. Ph.D., Purdue University, August 1994. Collaborative Multimedia Environments for Problem Solving. Major Professor: Chandrajit Bajaj.

A new method of designing collaborative multimedia environments for computer assisted problem solving is described. These environments support computer mediated interaction between multiple physically separated users joined by a communication network. Users interact using application specific models and objects, text, audio, video and graphics. Computer mediation enables both synchronous and asynchronous interaction, empowering users to transcend barriers of space and time.

Proliferation of high performance multimedia workstations and high speed and capacity networks provides us with the mechanism to realize real-time multi-user tools for computer-supported cooperative work. However, development and deployment of groupware, and consequent popular adoption, has been impeded by the absence of general models and enabling infrastructures. This thesis is a step towards developing formalisms for designing and implementing collaborative systems and groupware.

Requirements for the infrastructure from the developer's and the user's perspectives are identified and previous work is surveyed to highlight lessons learnt, and to isolate desired features that are lacking. Application models that are amenable to distributed and collaborative operation on heterogeneous platforms are then developed. In these models, software tools consist of contexts that are characterized by a state that is modified by events and can be thought of as event driven distributed data flow machines. These models are used to build an enabling infrastructure for rapid prototyping of real-time groupware. Mechanisms for routing events to different states

and contexts are provided, as are mechanisms for distribution functionality like synchronous and asynchronous remote procedure calling, and collaboration functionality like session control, interaction control, and high level access regulation. Identified shortcomings of extant work are overcome and mechanisms to implement policies derived from related research efforts are provided. The solution is justified from the technical and human factors viewpoints.

In this dissertation, the models and the infrastructure are described. Details of an implemented collaborative multimedia environment are presented, demonstrating the viability of the infrastructure. Possible applications of this technology are identified, and the facilitation of groupware prototyping by the model and infrastructure is described. Open issues and possible research directions are identified.

1. INTRODUCTION

Proliferation of high performance multimedia workstations and high speed and high capacity networks provides us with the mechanisms to realize real-time multi-user tools for computer supported cooperative work. However, development and deployment of groupware, and consequent popular adoption, has been impeded by the absence of an enabling infrastructure. This thesis is a step towards developing formalisms and models for designing and implementing collaborative systems and groupware.

We propose a new method of designing collaborative multimedia environments for computer assisted problem solving. These environments support computer mediated interaction between multiple users joined by a communication network. Computer mediation enables both synchronous and asynchronous interaction, enabling users to transcend barriers of space and time. Users interact using application specific models and objects, text, audio, video and graphics.

This section (Section 1) introduces essential concepts of Computer Supported Cooperative Work and surveys the state of the art, to put this thesis into context. We present requirements for a CSCW infrastructure from the application developer's and the user's standpoints. We survey previous work in groupware, multimedia and concurrent engineering, highlight lessons learnt, and identify desired features that are lacking. We then present an overview of the main features of the work described in this thesis.

In Section 2, we develop application models that are amenable to distributed and collaborative operation on heterogeneous platforms. The structural model proposes an architecture for developing software tools in a distributed multi-user setting. In this model, tools consist of contexts that are characterized by a state that is modified

by events. Tools readily interoperate with other tools. The media model allows integration of multiple media types into tools. The collaboration model enables groupware development. In the context of these models, multi-user applications can be thought of as event driven distributed data flow machines.

In Section 3 we describe how the models we propose can be used to implement an enabling infrastructure for rapid prototyping of real-time groupware. The infrastructure provides mechanisms for routing events to different states and contexts. It provides mechanisms for distribution functionality like remote procedure calling, and collaboration functionality like session control and high level interaction control and access regulation. It provides techniques for advanced interaction functionality, such as multimedia and graphics. The infrastructure overcomes identified shortcomings of extant work and provides mechanisms to implement policies derived from other related research efforts. We defend the infrastructure from the technical point of view of the application developer.

We describe the runtime system in Section 4. It consists of cooperating tools built upon the described models using the enabling infrastructure. We present details of the collaborative multimedia environment that we have implemented on the desktop. We discuss the viability and flexibility of the infrastructure and how it supports a heterogeneous mix of platforms. We describe collaborative tools and present cooperative interaction in different problem solving scenarios and applications. We highlight the relevance of computer-enhanced media-rich interaction for cooperative tasks. We defend the system from the human factors point of view of the application user.

Finally, in Section 5, we highlight the main features of this work. We identify other applications of this technology, and describe how the models and infrastructure facilitate rapid prototyping of sophisticated multi-user applications. We identify open issues and possible research directions.

1.1 Computer Supported Cooperative Work

CSCW refers to computer assisted coordinated activity carried out by a group of collaborating individuals. Groupware refers to multi-user software that supports CSCW systems. It is essentially the information technology that is used to help people work together. It often includes styles and practices of group process and dynamics that are essential for group activity.

Groupware and CSCW systems emphasize human-human coordination, communication and problem solving. By supporting audio and video communication and allowing work to be performed synchronously as well as asynchronously, they allow users to transcend the traditional requirement of being in the same place and working together at the same time.

In [44] Ellis *et al* review CSCW technology in depth, and introduce the major issues in that area. They define groupware as characterized by a common task performed in a shared environment. They present perspectives from distributed systems, communications, human-computer interaction, artificial intelligence, and social theory and discuss design issues like group interfaces and group processes. Bannon and Schmidt [19] discuss two main CSCW requirements – sharing an information space, and designing effective socio-technical systems, and assert that CSCW should aim at supporting self-organization of cooperative ensembles as opposed to disrupting cooperative work by computerizing formal procedures.

Researchers from behavioral science, sociology, management, and, of course, computer science have addressed issues in CSCW. In this section we attempt to glean lessons from the state of the art, in order to identify the main elements of collaborative work, and to point out requirements of an enabling infrastructure that makes it easy to build groupware.

1.1.1 Human Factors

Kraut *et al* stress the importance of informal, unplanned communication in scientific research [76]. They compare and contrast formal and informal communication. They discuss the surface characteristics of informal communication – frequency, method of initiation, location and duration. They discuss the content and uses of informal communication, including its perceived value for production and social function, its effect on personal perception, its role in sustaining momentum in collaborations, and its effect on the frequency of collaborative activity. Based on multiple interviews in collaborative work contexts, Bullen and Bennett [27] report that groupware and software tools that parallel non-electronic activity have great value in collaboration.

Clement [32] argues that the primary determinants of individual productivity are timely access to appropriate expertise and the ability of users to cooperate amongst themselves. He suggests that support for cooperative work and informal communication needs to be included in all modern computer systems.

Based on an ethnographic study, Nardi and Miller [94] assert that collaborative spreadsheet development is more a rule than an exception. Spreadsheets support sharing of programming expertise. The visual format for structuring and presenting data supports sharing of domain knowledge. In [106] Posner and Baecker report on how people write together based on interviews of many people involved in that line of work. They reveal the highly textured, multifaceted nature of collaborative activity. They examine participant roles, writing activities, writing strategies, and document control methods in the course of the process, and show how real collaboration flows smoothly between phases of brainstorming, planning, writing, and editing, and between synchronous and asynchronous activities.

Tang uses in-depth behavioral observations and video analysis of collaborative drawing to illustrate the design process [128]. He analyzes drawing space activity in terms of actions like listing, drawing and gesturing and functions like storing information, expressing ideas and mediating interaction. He emphasizes that gesture is

vital to effective collaborative interaction, and that the process of creating and using drawings conveys significant information not subsequently contained in the drawings. He shows that drawing space is an important resource for mediating collaboration.

Key concepts and theories of group and organizational process have been addressed by researchers. Flor and Hutchins [49] analyze the issue of distributed cognition – a new branch of cognitive science devoted to the study of representations of knowledge both inside and outside the heads of individuals. It deals with the propagation of knowledge among different individuals and the transformations that external structures undergo when operated on by individuals and artifacts. Distributed cognition helps us understand the behavior of teams of people engaged in complex cognitive tasks. It promotes reuse of system knowledge, and sharing of goals and plans, and provides shared memory for old plans and methodologies. It enables creation of efficient communication, and supports the ability to search through a larger space of alternatives. It supports joint production of ambiguous plan segments, division of labour, and specification of functional roles. Flor and Hutchins assert that a common problem in dealing with group processes is the failure to account for the complex cognitive processes in group problem solving.

Effective groupware can facilitate many kinds of group processes – discussing, planning, problem solving, writing, and designing etc. Behavioral and social studies show that successful groupware must proceed from an informed view of the dynamics of small groups, and that to be successful, even excellent groupware technology must be adopted and deployed with great sensitivity to the work context.

Ethno-methodology, the study of work cultures, and conversation analysis, the study of interaction, help understand and characterize group processes and groupware usage. Conversation analysis has become a vital tool for understanding the impact of groupware. Common ground in cooperative work refers to mutual knowledge, beliefs and assumptions. It is this grounding that contributes to collaboration.

Participatory design is a collaborative method for design of collaborative software. Studies indicate that for maximal benefit groupware design must be an iterative

process that is user-centered, and needs to involve multi-disciplinary input. Bullen and Bennett [27] call groupware social and technical intervention and assert that such tools are not used if the benefit derived from the technology does not outweigh the resources to be invested in using it.

In an investigation into lack of popularity of early groupware, Grudin [62] identifies factors responsible. He asserts that if users who make the most changes to adopt groupware do not receive the most benefits, or if the technology threatens existing socio-political structures, the technology will not be adopted. The same holds if groupware does not allow for a wide range of exception handling and improvisation, and if group enabled applications are hard to learn to use. He argues that successful groupware needs to have unobtrusive group work features, and must be skillfully introduced and deployed in order to reach the critical mass needed for popular adoption.

1.1.1.1 Lessons

Cooperation in tasks is an integral part of work, and involves both synchronous and asynchronous interaction. It is important to keep in mind social aspects of group dynamics when designing multi-user interfaces and group algorithms for computer mediated cooperation. Usability and ergonomic considerations are major factors for technology adoption. From the human factors viewpoint, it is useful to provide facilities for initiating and conducting informal communication for collaborative work. Drawing is an important aspect of communication, as is the associated gesturing and the very process of drawing creation. Cognition and awareness in the shared context streamlines the collaborative process.

1.1.2 Applications

In [73] Johansen identified some main applications that provided computer support for business teams.

- face to face meeting facilitation
- group decision support systems
- computer extensions to telephony
- presentation support software
- project management software
- calendar management for groups
- group authoring software
- computer supported face-to-face meetings
- computer screen sharing software
- computer conferencing systems
- text filtering software
- computer assisted audio-video conferences
- conversational structuring
- group memory management
- computer supported spontaneous interaction
- comprehensive work team support
- non-human participants in team meetings.

Instances of all but the last of these applications have been implemented and reported in the literature [50, 80, 134, 43]. Robinson [112] reviews and critiques classic first generation CSCW applications. This includes group authoring, calendar management, conversation management, work team support, group decision support, and spontaneous interaction. Group facilitation, a dynamic process involving managing

relationships between people, tasks and technology, and contributing to effective accomplishment of the outcomes of meetings is discussed in [43].

1.1.2.1 Lessons

The current trend of using todays powerful desktop workstations to mediate computer based communication and interaction gives incredible scope and breadth to the field of CSCW. It is likely that most software tools in the future will be group aware, as opposed to most current tools that attempt to preserve the illusion of being the sole user of a system.

1.1.3 Asynchronous Collaboration

Asynchronous groupware refers to group tools in which activity at the endpoints is completely delinked, and there is a possibly unbounded temporal separation between cooperative tasks. This class has achieved greatest prominence, and includes electronic mail, and computer conferencing in the form of electronic newsgroups and bulletin board systems.

Electronic mail is definitely the most successful form of groupware to date. Not only has it performed exceedingly well as a substitute for physical mail, it has also radically affected work culture. [124, 131, 123, 47, 46] report on various organizational, operational, and enabling effects of electronic mail and bulletin board systems as collaboration tools in the workplace.

A popular application of asynchronous groupware is in implementing intelligent agents that exploit mail message structure. Electronic mail is being extended by embedding intelligence that aids in the structuring, routing and filtering of messages. The motivation of these factors increases as more information is provided through the medium of electronic mail. They provide better methods of organizing, classifying and managing messages. One goal is the creation of intelligent messaging systems where specifiable tasks are delegated to computer processes. Applications are message-enabled by use of a store-and-forward messaging transport mechanism that moves

information from one person to another, and notifies participants in the process. This includes forms-routing, scheduling and calendar programs. Messages may also define, embody and manage workflows. [86, 78] describe environments for intelligent electronic mail management and CSCW systems based on them. These systems are similar to those in the intelligent office systems field, where researchers attempt to develop formal descriptions of office procedures and systems that embody these procedures.

Another application area is that of active mail, which involves sending active "agents" *via* electronic mail, that already has a well established infrastructure. Borenstein [21] describes the concept of computational electronic mail, defined as the embedding of programs within electronic messages. He discusses the promise of this technology, and key problems like security and portability. Goldberg *et al* propose active mail as a tool for maintaining persistent interactive connections and list applications of the concept [57].

1.1.3.1 Lessons

Asynchronous group work is an integral part of work culture. Computer mediated communication allows us to move large amounts of information quickly between multiple members of a group. It is important to maintain human and computer processable forms of this information, and to provide mechanisms to create, store, transmit, retransmit, organize, filter, and search through it.

1.1.4 Physical Meeting Support

This includes environments and software to support electronic meeting rooms and decision rooms within one physical space, and has been addressed by much research. Mantei [89] describes the Capture Lab, a computer based meeting room, and discusses different physical factors that affect the effectiveness and usability of the entire system. [100, 105, 99] study and analyze electronic meeting support system technology. Liveboard is a whiteboard size interactive electronic display used by speakers

for sketching, gesturing and presenting slides [45]. [135] argues that typing takes up too much cognitive capacity to participate fully in computer-based meetings for many people, and describes a pen-based meeting support tool.

1.1.4.1 Lessons

Physical spaces can always be constructed to optimize any kind of group interaction when collaborators are copresent. Computer based mechanisms can be used to augment the physical space. It is important to minimize the processing overhead of the computer based mechanisms in order to maximize participation and input, especially for real-time synchronous interaction.

1.1.5 Media Spaces

A media space is a computer controlled teleconferencing or videoconferencing system where audio and video are used to transcend physical barriers, to create shared interpersonal spaces across a distance. Media spaces can be used to link geographically separated collaborators. [77, 2, 90, 65] discuss some systems and issues. They demonstrate the use of video both as a viable alternative to face-to-face interaction, and as a means of sharing a workspace. However, though the sense of presence is conveyed, verbal and nonverbal cues are not transmitted as well as in a face-to-face situation. [28] stresses the role of gaze, body language and eye contact in shared "person space" – the collective sense of copresence between group participants. Dourish and Bly [42] argue that full bandwidth video is not absolutely necessary as there is much useful information even in low bandwidth video.

Heath and Luff [66] discuss social aspects of media spaces that have significantly different characteristics from shared physical spaces. TeamWorkStation [71] is an exploration into the potential payoff from special purpose hardware for visually combining displays of shared digital surfaces with the displays of physical work surfaces and materials. Clearboard [70] uses elaborate hardware and attempts to remove the

seam between shared personal space and shared task space, and provides smooth transitions between face to face conversations and shared drawing activities.

1.1.5.1 Lessons

Media spaces provide mechanisms to simulate physical copresence in the visual and audio sense. Though gesture and other forms of visual conduct are less effective in a media space than in face-to-face communication, the very availability of video and audio greatly enhances the quality of interaction. The utility of audio diminishes significantly with loss of quality. Video, however, is useful even at very low frame rates as it promotes awareness.

1.1.6 Desktop Conferencing

This area deals with using desktop computers and communication networks to support group activity. The Colab project [127] is one of the earliest demonstrations of a variety of synchronous multi-user interfaces. They describe brainstorming tools for small groups of physically colocated people, based on the WYSIWIS paradigm – what you see is what I see. Arguments in support of WYSIWIS are presented in [129]. [24] discusses the utility of desktop conferencing in providing "conversational props" to aid communication, and describe a shared whiteboard. Greif [61] discusses issues in designing desktop conferencing systems and group enabled applications *via* explicit asynchronous transmission of shared information.

Screen sharing is a simple mechanism that allows collaborative use of interactive software without modification. Here, the display of the program is distributed to multiple workstations. Sarin and Greif [116] discuss implementation issues for real-time conferencing systems. Greenberg reviews the history of screen sharing applications, and discusses technical problems that must be solved in order to achieve viable implementations [59]. MMConf [38] implements an alternative to screen sharing – window sharing, where users continue to work in their private workspace while

collaborating within the window that represents the public workspace. Issues in window sharing in the context of current windowing system technology are discussed in [82, 81]. SharedX [53] allows users to share existing X based applications by replicating the window interface. Matrix [72] is an infrastructure to make existing single-user systems collaborative, and supports synchronous and asynchronous work. XTV (X Teleconferencing and Viewing) [1] and COMIX [16] are other window sharing systems. CECED [37] and MObViews [63] are other desktop conferencing systems built using window sharing technologies.

Collaboration aware multi-user sketching and drawing systems are described in [60, 98, 133, 83]. SEPIA is a collaborative hypertext browser that allows individual and shared browsing [64]. Quilt [48] and PREP [96] are asynchronous collaborative editing and authoring tools. GROVE [44] and ShrEdit [41] are synchronous multi-user editors. ICICLE [26] is a multiuser tool for program source code inspection.

1.1.6.1 Lessons

Desktop conferencing is emerging as a powerful mechanism that supports collaborative work by enabling accessibility and sharability. The technology is applicable to numerous problem solving domains. Screen and window sharing provide a very simple means of cooperative use of existing desktop applications by multiple users. The primary advantage is that the user does not need to learn new systems. The disadvantage is that the application cannot benefit from the fact that there are multiple users since it is collaboration transparent. This permits a very limited form of shared interaction – Users must take turns interacting with the application, though everyone shares the view. Window sharing systems can implement different floor control strategies to regulate turn-taking. Collaboration-aware desktop conferencing systems are harder to implement, and can support more complex forms of shared interaction. Developers of such multiuser tools for different application areas have addressed and independently, albeit repetitively, solved the same core set of problems in domain

specific manners. This is due to the absence of general models and infrastructures that enable groupware development.

1.2 Enabling Technologies

Collaboration involves performing common tasks in a shared environment. In general, synchronous collaboration has more infrastructural demands than asynchronous collaboration. An infrastructure for groupware must provide and enable easy incorporation of the core technologies in order to promote development of collaborative systems. The technologies most critical for collaboration are

- shared data management
- concurrency control
- distribution
- session control
- interaction control
- coordination control
- multimedia and graphics
- user interfaces

An important requirement of an enabling infrastructure is that it provide the requisite mechanism, and also the flexibility to implement different application specific policies. These mechanisms should support both synchronous and asynchronous collaborations.

1.2.1 Shared Data Management

At the lowest level, the notion of a common task in a shared environment manifests itself as shared data that is manipulated by software tools. Sharing of data can be

implemented *via* database systems, hypertext webs, distributed object systems, or by application specific means.

An enabling infrastructure must a flexible data sharing mechanism. In addition to the core requirement of sharability, important issues are ease of integration, portability, heterogeneity, efficiency and flexibility.

1.2.1.1 Database Systems

Modern database systems, based on very mature technology, are ideal for implementing the data sharing substrate. They provide a high level of abstraction and provide concurrency control as well as access control. Distributed database systems provide replication and support regulated simultaneous manipulation of data. Performance of traditional disk-based database systems, however, is an issue, since all data accesses and updates must occur *via* disk.

1.2.1.2 Hypertext and Hypermedia

The hypertext concept enables the creation of complex webs of information and provides computer based mechanisms of navigating this structured recorded information space. Hypertext blurs the distinction between authors and readers, enabling a new kind of reading, writing, teaching and learning. Conklin [35] introduces hypertext, and describes its characteristics, building blocks and application areas. The flexibility and power of hypertext make it a foundation technology for groupware. Hypermedia extends the hypertext concept to include multimedia. Important applications include collaborative knowledge building [118, 117], computer supported education [79], asynchronous collaborative writing [96], organizational memory to record methodologies and procedures [34], and general information infrastructures like the World-Wide Web [20]. SEPIA, a collaborative hypertext browser that allows individual and shared browsing is described in [64].

The decentralized nature of the hypertext model lends itself very well to collaboration scenarios, since it supports flexibility and promotes shared access, especially

in the asynchronous setting. The challenge lies in being able to harness the great freedom in creating and navigating complex information webs for group work.

1.2.1.3 Distributed Object Systems

These systems provide very flexible mechanisms for maintaining portable data in multi-platform and multi-language settings. Shared data substrates built on these systems have the advantage of efficiency and flexibility. This is still an evolving technology.

1.2.2 Concurrency Control

Simultaneous multi-party interaction over shared data in a distributed setting can result in anomalous conditions and data inconsistency. Concurrency control mechanisms allow consistent simultaneous access and manipulation of shared data. A variety of locking and timestamp based techniques have been researched in the database community.

The infrastructure must provide flexible concurrency control mechanisms that fulfill the efficiency requirement for real-time concurrent interaction.

1.2.3 Distribution Control

Groupware relies on linking individual workstations using communication networks. Networking technology is at the core of CSCW. [30] presents an overview of this field. The need for control in the distributed system that underlies groupware is discussed in [113]. Distribution control provides mechanisms to interact with programs and users across a communication network.

Requirements at this level include convenient and flexible connection setup, and synchronous and asynchronous data transport. The transport mechanism is used to support communication using different media types, which may or may not tolerate losses. Unreliable data transport is more efficient than reliable communication. It is

therefore important for the infrastructure to offer both reliable and unreliable channels, and high bandwidth. The infrastructure must also deal with issues of platform, system and language heterogeneity.

1.2.4 Session Control

Collaboration control mechanisms regulate how multiple users assemble and interact over shared data. They regulate session setup and tear down, formation of collaborative groups, and dynamic inclusion and removal of participants. Different application domains have differing needs for methods of initiation and conduction of collaborative activity in a distributed setting.

The infrastructure must provide flexible collaboration control methods to initiate and terminate collaborative sessions, to join or leave ongoing sessions, and to invite participation in collaborative tasks.

1.2.5 Interaction Control

Interaction control mechanisms govern issues like floor control and interaction regulation. Different applications, as well as different usage scenarios of applications, require differing kinds of interaction.

The infrastructure must provide mechanisms that allow turn based interaction as well as simultaneous multi-party interaction. For turn-taking based mechanisms, it must provide flexible and intuitive mechanisms to implement protocols for requesting, taking and giving up turns.

1.2.6 Coordination Control

Coordination is the act of managing interdependencies between different activities performed to achieve a goal. Coordination is necessary for rapid progress towards targets. Coordination theory is a body of principles about how people can work together harmoniously. The importance of coordination for group activity is discussed

in [87]. Coordination, though very domain specific, is achieved *via* the promotion of awareness of group activity and is enforced via access control over shared contexts.

1.2.6.1 Awareness

Awareness of individual and group activities is an important issue when performing collaborative tasks. Awareness is fundamental to coordination of activities and sharing of information, which is critical to successful collaboration. Sharing the character of activity allows users to structure their tasks to avoid duplication. Awareness of content allows fine-grained shared working. A study of awareness and coordination in collaborative activity is available in [41]. The authors define awareness as "an understanding of the activity of others, which provides a context for your own activity," and assert that it is especially important in semi-synchronous shared workspaces in the form of past shared activity.

The infrastructure must provide computer mediated mechanisms for promotion of awareness in a collaborative setting. This includes active and passive methods. In active methods users explicitly provide information about their tasks. In passive methods the system automatically collects and disseminates background information that conveys remote presence and the notion of remote activity *via* shared feedback presented in a shared workspace.

1.2.6.2 Coupling

Coupling refers to the degree of connectedness between collaborating interfaces as perceived by the user. At one extreme is WYSIWIS – what you see is what I see. In this setup there is maximal coherence between the views of shared activity that are available at all sites. At the other extreme is the scenario of totally decoupled views, and that of asynchronous interaction. Different users can have completely different views. Coupling of state refers to the connectedness of content of shared activity. Coupling of interaction deals with synchronicity of shared user interaction. Though the general requirement for CSCW is the maintenance of coherent shared state, [93]

argues for "lazy" consistency in distributed settings for coarse grained activity. A general framework for undoing actions in collaborative systems that allows users to reverse their own changes is presented in [107]. [40] discusses the notion of flexible coupling.

An infrastructure must provide flexible coupling mechanisms. This would allow application developers to implement coupling policies based on efficiency and performance considerations. It would also allow users to control the degree of synchronicity of state and interaction.

1.2.6.3 Access Control

Access control is a critical issue, since it provides control over what tasks can be performed by which individual in a collaborative setting. It provides the mechanism for enforcement of coordination, enables division of labor, and provides access regulation over shared state and interaction. Dewan and Shen [121] argue that "user interaction with a collaborative application can be interpreted as concurrent editing of data structures of the session". They develop a general access control model based on read, write, viewing, coupling and domain-specific rights.

The infrastructure must provide a flexible access control mechanism that allows application developers to implement specific policies, and users to dynamically control shared task progress.

1.2.7 Multimedia and Graphics

Communication is at the core of collaborative effort. A media-rich communication substrate greatly facilitates information sharing. Audio support is very useful for collaboration and video promotes awareness in this scenario. Graphics provides visual realism for many application domains.

The critical issues that need to be addressed by an infrastructure are support for platform heterogeneity, support for communication, and mechanisms for incorporating multimedia and graphics facilities to create sophisticated applications.

1.2.7.1 Multimedia

Rice and Steinfield [111] characterize communication media along the dimensions of constraints, bandwidth, interaction and network factors, and discuss issues in asynchronous multimedia communication. An overview of advances in interactive digital multimedia systems is presented in [51]. The promise of multimedia as an enabling technology for computer supported cooperative work is discussed in [25].

In [69], Hollan and Stornetta conjecture that communication *via* electronic media that imitates face-to-face communication can never achieve the social presence and media richness of a physical setting. However, computer mediation of this communication affords us added richness in terms of asynchronicity, archivability and reviewability – features that go beyond what the physical setting offers. Gaver [54] shows how non-speech audio can assist in cooperative work by helping to maintain common awareness. Auditory cues enable a relatively unconscious awareness of ongoing events and effectively enrich shared spaces by reinserting cues lost due to the absence of face-to-face interaction. Borenstein and Thyberg [23] describe the Andrew Message System that has multimedia mail capabilities and active messages for user support in a distributed computing environment.

Developers of multimedia tools and applications deal with a complex environment due to the variety of media and supporting equipment. Audio and video networking has made it possible to build distributed multimedia applications with multiple concurrent users, requiring real-time responses and dealing with multiple data streams. For rapid prototyping in the context of distributed multimedia applications, we need to identify general abstractions found in multimedia applications and integrate them into a framework that provides basic functionality and media integration mechanisms, promoting development through reuse.

1.2.7.2 Graphics

The quest for visual realism in computer based interaction has resulted in significant advances in graphics technology. Standards like PEX, PHIGS and OpenGL have

evolved. High speed networking has made it possible to build distributed graphics applications with multiple concurrent users. This enables sophisticated interaction functionality like shared virtual worlds.

Once again, developers of 3D graphics tools and applications deal with a complex environment due to the variety of hardware graphics platforms available. For rapid prototyping in the context of distributed graphics applications, we need to identify general abstractions found in graphics applications and integrate them into a framework that provides basic functionality and graphics integration mechanisms, promoting development through reuse.

1.2.8 User Interfaces

The user interface is the mechanism that ultimately expresses the sharing and cooperation paradigm. Groupware needs to support the notion of private and shared work in private and shared workspaces, and methods for moving work between these workspaces. The user interface must flexibly support customization and coupling, and must serve as a medium for expressing feedback to promote awareness and coordination. It must provide intuitive methods for session, interaction, and coordination control, and support media-rich communication.

The infrastructure must provide mechanisms for building such distributed and collaborative user interfaces. Interface design criteria and policies, and ergonomic issues should be addressed *via* participatory design to effectively capture multi-user processes.

1.2.9 General Requirements

In order to be maximally effective, groupware needs to bridge the traditional gaps between

- individual and group work and process
- work with conventional software and groupware

- work in private and shared space, in local and distributed settings
- synchronous and asynchronous work

In order to support design and deployment of such groupware, the enabling infrastructure must provide a rich set of flexible mechanisms for the described requirements.

1.3 Related Work

1.3.1 Groupware

Groupware refers to multi-user software that enables computer supported cooperative work. It focuses on using the computer to facilitate human interaction for problem solving. Ellis *et al* present an overview of the state of the art, and identify the main issues in this area that is centered around performing common tasks in a shared environment [44].

There are three traditional approaches to developing groupware. In the centralized collaboration-transparent approach, there is one instance of a single user software tool that is shared by multiple users by means of an underlying screen or window sharing mechanism. *E.g.* systems like SharedX [53], XTV [1] and COMIX [16] intercept the X [119] event stream and simultaneously drive windows on multiple displays. This approach allows users to share existing X based applications by replicating their window interfaces.

Lauwers *et al* [81] claimed that existing window managers are not well suited to supporting groupware. They suggested that changes would be required of window systems to support adequate spontaneous interactions, shared workspace management, floor control, and annotation and telepointing in collaboration transparent applications. MMConf [38], Matrix [72], MONET [125], CECED [37], BERKOM [6] and MObViews [63] are desktop conferencing systems built using window sharing technologies. The main advantage of the centralized collaboration-transparent approach

is that it eases groupware generation, since applications do not have to be changed. Users continue to employ familiar single-user software tools for group work.

There are many disadvantages to this approach. Users are forced to take turns, since the tools are not designed for multi-user interaction. The turn-taking mechanism itself is part of the window sharing system, and is thus external to the tool. Tools do not support inter-user interaction or communication, and all users are forced to have identical views. In a heterogeneous distributed environment tools need to operate on a greatest common denominator platform, and cannot take advantage of machine specific features like hardware graphics facilities etc. There is a lot of network traffic generated since all events must travel to and from the central tool. Centralized view generation in the shared tool and window sharing system does not scale well as the number of users increases.

In the centralized collaboration-aware approach, there is one instance of a multi-user software tool that drives multiple interfaces and is thus shared by multiple users. Systems like Rendezvous [104] and Weasel [58] provide mechanisms to implement this approach. A familiar tool that adopts this technique is Wscrawl [133]. The advantage of this approach is that it enables tools to implement floor control mechanisms internally, and allows multiple users to interact concurrently. Inter-user communication facilities can be provided, and it is possible to support different views and user customization. Centralization of the handling of collaborative interaction eases concurrency control.

However, there are disadvantages to this approach. In a heterogeneous distributed environment such tools either operate on a greatest common denominator platform, or are burdened with the complexity of taking advantage of machine specific features. There is a lot of network traffic generated since all events must travel to and from the central tool. There is a performance penalty for every additional user of the tool since the central tool does all view generation, and this does not scale well as the number of users increases. Both centralized approaches are susceptible to distribution issues

like network delays and throughput. They are also less robust, since the state of the shared task is centralized in one tool.

In the replicated collaboration-aware approach there are multiple instances of tools in the distributed environment. Each maintains a local interface and provides access to the shared task. The multiple tools cooperate to maintain the notion of shared state and interaction. [81] is a deep analysis of serious implementation challenges that must be tackled to keep copies of shared synchronous applications running under a replicated architecture synchronized with one another. It is asserted that to do this, one must guarantee input consistency, output consistency and startup consistency for the applications. One solution is to make some system components like the underlying window managers collaboration aware.

MMConf [38], LIZA [55] and GroupKit [114] provide facilities to implement replicated groupware systems. Rapport [4] and Diamond [38] are systems that implement this approach. The advantage of the replicated collaboration-aware approach is that it enables tools to implement floor control mechanisms internally, and allows multiple users to interact concurrently. Inter-user communication facilities can be provided, and it is possible to support different views and user customization. Tools can be built to operate on multiple platforms, taking advantage of available facilities, and sharing can be implemented in a heterogeneous environment. Since the notion of a shared task is maintained in a replicated distributed system, this approach is robust.

The disadvantage of this approach is that tools are burdened with the complexity of maintaining shared state and interaction in a replicated setting. Concurrency control is harder. Most importantly, scalability of performance becomes an issue when the number of users in the shared space increases. This is because a larger number of sites need to be kept in sync for fine grained shared interaction. For coarse-grained sharing, however, performance is better. Network traffic is reduced since tools can perform functions locally. Ahuja *et al* [5] present a comparison of architecturally different versions of the Rapport desktop multimedia conferencing

system. They discuss performance issues and recommend the single site centralized approach.

In addition to investigation of operating system issues for supporting groupware, recent research effort has been directed towards the design and construction of toolkits and languages for building groupware. The advantage of a language based approach is that it enforces formalisms, and enables automatic generation. The problem with this approach is the implicit requirement that shared tools be implemented in that language. This imposes many artificial restrictions on application development from the point of view of the underlying user interface system, graphics system, multimedia system, networking and communication system and implementation platform. This effectively works against widespread adoption and deployment of language based mechanisms.

Language based approaches to generating multi-user applications are described in [68], where Hill presents the Rendezvous collaborative user interface development environment. The Rendezvous system proposes an architecture for multi-user applications [104]. It implements a centralized collaboration-aware approach. The authors identify three dimensions of programming complexity that seriously affect multiuser applications – concurrency, that enables parallel activity, abstraction, that separates interface from underlying application, and roles, that address the need to provide different interfaces to different users. The Rendezvous language extends Common Lisp to support objects, message passing, event handling, graphics and constraint maintenance. The paper discusses the concepts and implementation experience in detail. The centralized approach, however, has inherent problems of scalability and performance.

Dewan and Choudhary [40] discuss the Suite system that provides primitives for programming multi-user interfaces. Suite is a language and system for developing both collaboration transparent and collaboration aware multiuser programs. Concurrent tasks can be implemented by a set of communicating distributed objects. Suite develops the notion of active variables, attributes and value groups, and supports

multi-user objects and a multi-user user interface management system. The authors describe flexible coupling that determines which user actions are seen by other users, and when they are seen. They discuss implementation experience and present ideas about applying the approach to other system and language contexts. On similar lines, Oval [88] is a tailorable tool for cooperative work. Users create applications using Objects, Views, Agents and Links. Objects represent data, Views summarize collections of objects and allow editing, Agents perform active tasks for users, and Links represent relationships between objects.

Weasel is another system for implementing multi-user applications [58]. It implements the relational view model – a user interface is described as a relation between a program's data structures and the view on a display. Users manipulate views of the data that are bound to application programs *via* relations. The Weasel architecture has multiple client views controlled by a central server. Views are specified in RVL, a declarative language. Weasel creates a distributed implementation from the specification, hiding network communication, concurrency control, synchronization and customization. The centralized server has inherent problems of scalability and performance.

CB (Conversation Builder) [74] is a support tool that provides active support for collaborative work activities. It assumes coarse grained collaboration, where users work independently on actual tasks and periodically synchronize their independent activity by resolving dependencies. CB Protocols allow different activity types and policies to be defined to the system. Obligations provide a mechanism to weave individual activities together. CB allows users to be aware of activities engaged in, the relations among activities, legal actions in an activity, and relevant actions of co-workers.

GroupKit presents a mechanism for creation of real-time work surfaces that are essentially shared visual environments [114]. It is structured around an extensible object oriented runtime environment that manages distributed processes and interprocess communication. GroupKit uses transparent window overlays to create shared

work surfaces, and supports open protocols for creation of interface and interaction policies. It supports gesturing and graphical annotation.

In [65], Harrison and Minneman investigate the use of media spaces as design tools. They characterize design as the creation of experiences, fundamentally a social activity, and assert that it is characterized by ambiguous communication, continual negotiations, and the enrollment of participants into a group process. They argue that video can help designers connect across space – through transmission over a network and across time – through recording and review. They define the concept of a media space, and review case studies of PARC media spaces. They show that designers can learn quickly to make effective use of video both as a viable alternative to face-to-face interaction, and as a means of sharing a workspace. Reeves and Shipman [110] propose a method for integration of the design of an artifact, which is the target of a task, and communication between designers. They assert that discussions about the design must be embedded in the design, integrated in a manner that provides a seamless environment for individual and group work.

Teledesign [122] is an application of synchronous groupware in 3D Computer Aided Design. The authors report experiences with a two-person replicated design tool from the perspectives of simultaneous versus turn based access, and degree of sharing. They posit that two-person meetings do not need a moderator, simultaneous editing is not chaotic, and telepointers are useful, as are visual cues of remote viewing position.

1.3.2 Multimedia

In the direction of shared multimedia environments, research in colocation has resulted in systems like MONET [125], MMConf [38], Rapport [4], CECED [37], and MERMAID [132]. These systems primarily provide audio-video communication. Some of these systems also provide conference management facilities and content independent sharing of drawing and viewing surfaces.

A paradigm for modeling multimedia collaborations and their system requirements is presented in [109]. The authors propose a three-level hierarchy. Streams consist

of media communication modulated by access rights within a collaboration. Sessions are collections of semantically related streams. Conferences are temporally related sequences of sessions. They assert that a common software framework with rich semantic expressibility is essential to support the diverse range of interactions required for synchronous and asynchronous collaboration.

AudioFile [84] is a network-transparent system for distributed computer audio applications built using lessons from X [119]. It provides an abstract audio device interface *via* a simple network protocol. AudioFile is a step towards systems that deal with media at an abstract, semantic level.

KWrite [52] is a system based on the Apple Macintosh System 7 Inter Application Protocol. The authors describe an open architecture for multimedia documents that offers the possibility that any application that interacts with a user through a window can also interact with the user through an active picture in such documents. This enables interactive applications to use the document as a user interface while appearing seamlessly embedded to the user.

Gibbs [56] proposes the notion of an active multimedia object that has the autonomous ability to send multimedia data to outside entities like screens and networks. Gibbs and Mey [92] propose a method for rapid prototyping of multimedia applications. They include general abstractions of multimedia applications into an extensible set of related classes that provide basic functionality and composition mechanisms. They adopt a component-oriented view, using visual tools for constructing and configuring applications.

Traditional work in distributed multimedia systems has focused on transmission, synchronization, and operating system support for continuous media streams. Integrated control of remote multimedia devices like cameras and speakers is addressed in [75]. The authors discuss an application level architecture, and a protocol for control of external devices. The VidBoard [3] is a standalone network based video capture and processing peripheral capable of capturing and transmitting live television source. The system is described in detail.

Electronic mail is the most pervasive groupware technology to date. Networking and communication technology have been effectively applied to allow creation, storage, transmission and retransmission of messages. MIME (Multipurpose Internet Mail Extensions) [22] proposes a method for extending and using the existing mail infrastructure to enable a richer form of asynchronous messaging. The method leaves the message content as flat ASCII text. It redefines the format of message bodies to allow multi-part textual and non-textual message bodies to be represented and exchanged without loss of information. It provides facilities to include multiple objects in a single message, to represent body text in character sets other than US-ASCII, to represent formatted multi-font text messages, to represent non-textual material such as images and audio fragments, and generally to facilitate later extensions defining new types of Internet mail for use by cooperating mail agents.

MHEG [108] is an upcoming standard for hypermedia object interchange. Its objective is to address "the coded representation of final form multimedia and hypermedia objects that will be interchanged across services and applications by any means like storage media, local area networks, and wide area telecommunication and broadcast networks." The MHEG Object is the basic component, and is intended to play a federating role, enabling different applications to share the basic information resource. HyTime (Hypermedia/Time-Based Structuring Language) [97] is a standard that specifies how concepts considered universal to all hypermedia documents can be represented using SGML (Standard Generalized Markup Language) [31]. This allows hyperdocuments to be represented as character files that can be interchanged between and processed by any platform.

1.3.3 Concurrent Engineering

Concurrent Engineering is an applied area of Computer Supported Cooperative Work that centers around methodologies and tools that enable cooperative decision

making by geographically distributed people engaged in all aspects of product development. The primary issues dealt with are colocation, information sharing, integration, coordination, and corporate memory. Collocation provides virtual copresence *via* media-enhanced communication. Information sharing deals with aspects of actual sharing of artifacts and targets of design and design process. Integration deals with inter-operation of different tools and techniques used in the design process. Coordination deals with mechanisms to keep track of team progress, and to regulate team activity. Corporate memory is concerned with capture and use of decision rationale. We are specifically interested in the areas of colocation, information sharing, integration and coordination.

DICE (DARPA Initiative in Concurrent Engineering) contains many projects that emphasize the combination and reuse of existing heterogeneous tools. Tools use wrappers to communicate in a mutually understood language, protocol and representation. PACT (Palo Alto Collaborative Testbed) [39] addresses the problem of linking existing collaborative engineering environments, to enable their use in other projects. Its architecture encapsulates functionality in agents. Facilitators are used to link agents across environment boundaries, using a standard language to communicate between environments.

SWIFT [85] is a computer environment under development that is aimed at enhancing group problem-solving productivity. It consists of a Knowledge Layer and a Kernel Layer that underly a Collaboration Layer being built to enable rapid application development by retargeting existing functionality.

DICE (Distributed and Integrated Environment for Computer-Aided Engineering) [126] is centered around a persistent shared blackboard implemented by a global object oriented database. It contains negotiation, coordination and solution components. Knowledge modules interact with the blackboard and are responsible for translation of representation formats. A control mechanism evaluates and propagates results of action by message passing between knowledge modules.

The SHARE Project [130] is a concurrent engineering environment directed towards applying information technologies to help design teams gather, organize, access and communicate design information. It is being built around existing software tools, multimedia-enhanced electronic mail, window sharing methods for sharing applications between multiple users, and mail-based tool interoperation using ServiceMail.

CORBA (Common Object Request Broker Architecture) [103] and DCE (Distributed Computing Environment) [115] are standardizing distributed systems and enabling cross platform and cross language communication. Research effort (*cf.* [39]) has resulted in technologies like EXPRESS – a language for describing information models, PDES – Product Data Exchange Standard, KIF – Knowledge Interchange Format, and KQML – Knowledge Query and Manipulation Language. They enable cross-discipline and cross-application communication of information. gIBIS [36] – a graphical issue based information system, and DRIM – Design Recommendation-Intent Model, provide methods to capture and express design rationale.

1.4 Motivation

Computer systems have evolved from single user to time-shared multi-user systems. Traditional database systems and file systems allow sharing of data, while attempting to present to the system user the illusion of isolation. Research in Computer Supported Cooperative Work has entailed a paradigm shift, enabling users to be aware of others using the system, as well as interacting with them. This has extended the notion of sharing beyond simple sharing of data to sharing of computation.

Most current systems for CSCW and concurrent engineering are built on top of general technologies like databases and shared window systems to support information sharing. Systems that provide content independent sharing support concurrent access *via* serialized interaction. They can support only coarse grained concurrency, since they are not cognizant of the structure of the actual information being shared. Therefore, they provide limited flexibility in controlling the degree of sharing, and in the actual sharing.

As described earlier in this section, many of the underlying technologies are well understood, and are the focus of much research. Proliferation of high performance multimedia workstations and high speed and capacity networks, coupled with other support technologies, provides us with the mechanism to realize real-time multi-user tools for computer supported cooperative work. However, development and deployment of groupware, and consequent popular adoption, has been impeded by the absence of general models and enabling infrastructures. Groupware developers have to deal with the difficult task of marrying these multiple technologies due to the absence of high level semantic models that relate them, and infrastructures that ease the task of using them. The requirement of operating in a heterogeneous distributed setting further compounds the problem.

Our aim is threefold. We attempt to define high level semantic models for tools, interaction, and sharing. We also attempt to create an infrastructure that understands the core underlying technologies, and provides abstractions that enable application developers to build groupware. The abstractions stress on semantic level handling hiding actual details of lower level implementation. We accept and acknowledge heterogeneity in the real world, and capture and encapsulate it in the abstractions. Finally, we attempt to build multi-user tools and collaborative problem solving environments using the models and infrastructure.

A very central theme is that of openness and extensibility. It is unlikely that any specific software tool will ever encompass all the functionality that a user might reasonably require. We propose an open architecture tool model that supports integration with independent tools. The model provides cooperation *via* interoperation. It has a highly generalized architecture for integrating a heterogeneous range of information technologies. Interoperation allows function and content of any tool to be accessed by another tool. Various tools can be cross coupled and linked in a variety of interactive ways. We develop a media model for interaction in which any form of structured data with defined interaction semantics is treated as a media type. This model enables integration of audio, video, 2D and 3D graphics, and text into tools,

and extends to support application specific objects, spreadsheets, databases, animations, simulations, and hypertext and hypermedia. We develop a sharing model that extends the content and function sharing of interoperation by providing mechanisms to control and regulate synchronous and asynchronous shared interaction.

Reviewing the core technology requirements of CSCW infrastructures, shared data management tends to be domain dependent, and can be implemented on any of the mentioned technologies, or by using combinations of those technologies. Existing systems tend to use domain specific methods. Concurrency control is a mature field, and well known techniques exist. It is closely tied in to the data sharing model. We do not propose any new ideas in these areas. Coordination control is inherently domain and task specific, and we do not attempt to specify general models and techniques. It can be implemented on top of an effective communication infrastructure.

However, high level abstractions for the following areas are inadequate in the state of the art

- Distribution Control
- Collaboration Control
- Multimedia
- Graphics
- User Interfaces

We present an infrastructure that attempts to fill the gaps in order to support virtual spaces for flexible collaborative interaction. The infrastructure lets us build tools with shared drawing and viewing surfaces by supporting content dependent sharing – the tools are collaboration aware, and support synchronous multi-user manipulations of application-specific objects. This adds a new dimension to the kind of cooperation that can occur in collaborative problem solving, because it permits cooperative browsing of objects and interaction in the context of tools that manipulate those objects. Since tools understand the structure of the data they manipulate, this allows a

great degree of flexibility in sharing and concurrency control. It supports cooperation in the design and problem-solving phase, as well as in the review and analysis phase.

1.5 Highlights

We describe the main features of our groupware enabling infrastructure that separate it from related work. We also introduce how we have used it to build a collaborative multimedia environment for problem solving.

1.5.1 Structural Model

We have developed an architectural model for distributed and collaborative tools that emphasizes the separation of interface and function. In this model tools consist of "contexts" (views) that are characterized by "state" that is modified by "events". Contexts may be local or remote. State may be private or shared. Events may be user "actions" or "triggers". Events affect the private or shared state and can cause multiple local and remote contexts to be altered simultaneously, synchronously or asynchronously. The tool can be thought of as an event driven data flow machine that has mechanisms for routing events to different states and contexts. Distributed and collaborative tools are built by setting up the appropriate state and contexts, and by describing how events alter them. This model is described in Section 2.1.

1.5.2 Media Model

We propose a model for media enabled tools. Any form of structured data with well defined interaction semantics is treated as media. In this model tools interact with media "agents" that receive input from "sources", apply "filters" to the media stream, and generate output to "sinks". In conjunction with the Structural Model, this enables multimodal user interaction, distributed interoperation, and synchronous and asynchronous conferencing. This model is described in Section 2.2.

1.5.3 Collaboration Model

We propose a model for collaboration based on the Structural and Media Models. This is a flexible collaboration model that supports media-enhanced synchronous and asynchronous multi-user interaction. The model can implement traditional centralized and replicated collaborative tools, and also supports a new Session Model for collaboration, that allows for persistence and asynchronous interaction. This model is described in Section 2.3.

1.5.4 Distribution Substrate

This fulfills the need for distribution control, and provides a mechanism to implement shared data management for CSCW. It enables client-server and peer-peer interaction. The substrate provides mechanisms of setting up connections across the network, and flexibly managing data in a distributed setting. It provides device independent data transport for heterogeneous environments. It implements synchronous and asynchronous remote procedure calling and provides multiple-connection management between instances of tools. It supports several application level communication protocols. This substrate is described in Section 3.2.1.

1.5.5 Collaboration Substrate

This fulfills the need for Collaboration control and provides mechanism for interaction control and access regulation. It enables multi-user interaction. The substrate uses the distribution substrate to implement shared state and context in a distributed setting. It provides session management, interaction control and access regulation facilities that enable rapid prototyping and development of collaborative tools and groupware. This substrate is described in Section 3.2.2.

1.5.6 Portable Graphics

This is an abstract 3D graphics system that lets us access hardware graphics facilities of workstations in a device-independent manner, by presenting a high level interface to 3D graphics. It provides source code level compatibility across different graphics platforms in a heterogeneous setting, by implementing a hardware independent graphics library. It deals with the issue of heterogeneity for CSCW. It is described in Section 3.2.3.

1.5.7 Collaborative Graphics Substrate

It is based on the Structural Model and uses the distribution, collaboration and graphics substrates to implement device independent distributed and collaborative graphics. It supports synchronous and asynchronous 2D and 3D graphical interaction in a heterogeneous setting. It enables incorporation of graphics facilities into tools. It provides high level control of display and visualization parameters and supports telepointing. This substrate is described in Section 3.2.4.

1.5.8 Portable Multimedia

This abstract multimedia system provides access to available hardware audio and video facilities on a workstation in a device-independent manner, providing source code level compatibility across multiple platforms. It encapsulates details of media format and device specific interaction, providing a high level abstraction for development of multimedia tools. It deals with the issue of heterogeneity for CSCW. It is described in Section 3.2.5.

1.5.9 Collaborative Multimedia Substrate

It is based on the Structural Model and uses the distribution, collaboration and multimedia substrates to implement device independent distributed and collaborative multimedia. It enables incorporating multimedia features and facilities into tools, and

supports collaborative multimedia interaction. This substrate is described in Section 3.2.6.

1.5.10 Collaborative Applications

Sha-Draw and Sha-Poly are collaborative graphics tools. They are described in Appendix A. Sha-Phone, Sha-Video, and Sha-Talk are multimedia conferencing tools that have been implemented. They are described in Appendix B. Sha-Chess is the implementation of a virtual chess board that supports synchronous multi-user interaction in a distributed setting. It is described in Appendix D. Shilp is a solid modeling toolkit that supports synchronous participatory collaborative design. It is built using the media-rich substrates of the Shastra environment (Shastra is the Sanskrit word for a branch of knowledge or a science.) It is described in Appendix C.

We describe Shastra, a collaborative multimedia environment, and some problem solving scenarios in Section 4. The environment for collaborative geometric design is described in [8, 10]. The environment for collaborative custom design of artificial implants for human limbs is described in [14]. It uses the distribution and multimedia conferencing facilities of Shastra in conjunction with scientific design and manipulation tools. A distributed and collaborative volume visualization environment is described in [15].

2. MODELS

Building collaborative environments is not a very straightforward task because of the large number of factors that need to be taken into consideration. The process is made all the more daunting if media-enhanced tools are to be created. However, developing a foundational model and expressing it in terms of formulated abstractions can greatly ease the task of building such systems.

In this chapter we propose a model for the structure of tools that makes them amenable to collaborative multimedia interaction. We also propose a model for incorporating multiple media facilities into tools that emphasizes inter-operation. This eases the task of building tools since the developer can build on top of high level abstractions that implement much functionality. Also, tools based on these models can very easily be group enabled, and support collaborative media-rich user interaction. Finally, we propose a collaboration model based on the structural and media models. This model provides an infrastructure for building collaborative multimedia environments for problem solving.

2.1 Structural Model

We have developed an architectural model amenable to distributed and collaborative tools. The model emphasizes the separation of interface and function. Tools are the building blocks of distributed and collaborative environments. In this model tools consist of "Contexts" that are characterized by "State" that is modified by "Events" using the functionality in the "Core" *via* a dispatch mechanism, the "Mapper".

The model is depicted in Figure 2.1. The Application Core implements actual data manipulation functionality. Applications consist of possibly multiple Contexts, which may be local or remote. State may be private or shared. Events may be user

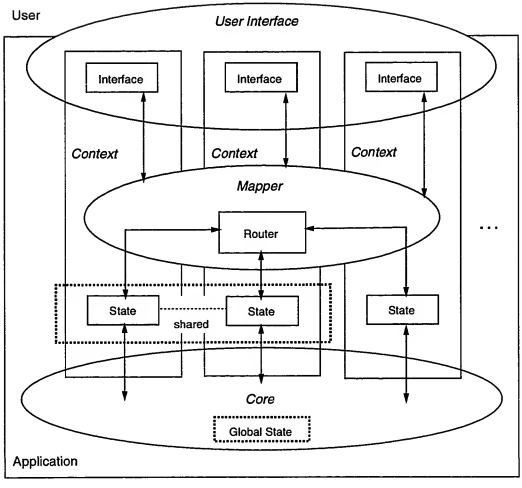

Figure 2.1 Structural Application Model

actions or synthetic triggers. The tool can be thought of as an event driven data flow machine that has mechanisms for routing Events to different States and Contexts.

Distributed and collaborative tools are built by setting up the appropriate States and Contexts, and by describing how Events alter them. Such tools operate on top of connection and transport mechanisms that are orthogonal to the tool model. Data sharing for collaboration is implemented by mechanisms that are also orthogonal to the model.

2.1.1 Core

The Core of a tool is the basic set of functions that it provides as a usable system. The Core uses state information and user input to respond to the user, alter state information, and produce output.

2.1.2 Contexts

A Context is essentially a view of the state of a tool, and the data it is manipulating. It is also the mechanism of expressing user interaction. It provides an Interface, usually a GUI, *via* which the user interacts with a tool and accesses its functionality. Applications can consist of multiple independent or dependent contexts. Dependent Contexts allow the user to maintain different views of the same shared State data. Independent Contexts contain unrelated State data. All Contexts utilize the tool Core to manipulate the data. Contexts present the results of manipulation through their Interfaces. A Context is associated with a unique identifier that serves as an address.

The concept of Context is policy free. Contexts may be Local, expressing results of local user interaction. Alternately, they may be Remote, expressing results of remote interaction. Applications may disallow user interaction with Remote Contexts, using it only to express remote state information. Or they can allow the user to interact with Remote Contexts for shared interaction.

2.1.3 Interfaces

The Interface is the medium *via* which a tool Context expresses itself. The Interface concept is policy free. Different Contexts may share the same actual user interface. In this case the user interface of the tool would typically provide methods of switching between different Contexts. Also, Contexts would be responsible for correctly displaying the State information in the shared interface. Alternately, Contexts may have physically separate user interfaces. The actual implementation of an Interface is dictated by domain and tool specific requirements.

2.1.4 State

State of a Context is essentially the data that is being manipulated by the tool in that Context, as well as meta-information about how the data is translated to a view. The tool operates by performing actions on data in a Context and expressing results *via* the corresponding Interface. State may be private or shared. Private State is the usual notion of data manipulated by a tool. Independent Contexts in a tool have private State. Private State can be manipulated only by local Events. Dependent Contexts have shared State. Data in a shared State may be manipulated by Events of local and remote origin.

2.1.5 Events

An Event is the unit of user interaction with a Context of a tool. Tools perform actions on data in response to events. Our notion of Event is at a high level of abstraction, and is policy free. Tools may maintain the notion of Events at as low a level as key strokes or user interface management system and windowing system events. Alternately, they may maintain the high level notion of tool actions.

In this model, Events may be user Actions or Triggers. User Actions represent actual interaction sequences that cause the tool to perform an operation on Context data. Triggers are synthetic events in the sense that they are not initiated directly

by a user. They are generated as a result of the operations performed by a tool due to an Event.

Events identify the Context of origin, and the Context they affect. They cause modifications in Context State. Events that affect shared State can cause multiple local and remote Contexts to be altered simultaneously, synchronously or asynchronously. This is achieved by sending messages to the other Contexts.

2.1.6 Mapper

The Mapper of a Tool is a dispatch mechanism for Events. It is built on top of a messaging system. The messaging system implements a router. The messaging system and routing mechanism is orthogonal to the Structural Model, and is largely transparent to the Mapper. For Events that affect a local Context, the Mapper invokes functionality embedded in the Core. For Events that affect remote Contexts, the messaging system lets the Mapper direct events to remote Contexts in the form of messages.

2.1.7 Messaging

Messaging is the mechanism by which Contexts communicate. Contexts communicate with the Core *via* the messaging subsystem in the Mapper to cause it to perform actions in response to Events. Contexts communicate with each other to maintain shared State information. Shared State in a Context is set up to generate Trigger Events whenever its data is altered. The location of generation of these Triggers is significant. If Triggers are generated before state is altered, we we can achieve input replication. If triggers are generated after state is altered, we can achieve output replication. Every Context is associated with a unique identifier that serves as an address to which messages are directed.

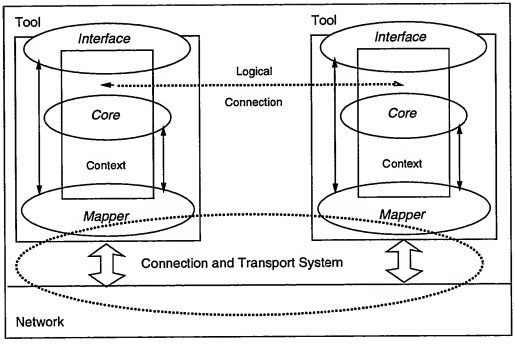

Figure 2.2 Distribution Model

2.1.8 Routing

The messaging subsystem maintains information for all Contexts, local and remote. It uses the connection and transport mechanisms that underly the distributed architecture to acquire remote information and transmit local addressing information. It implements a distributed messaging system *via* which messages are routed to the appropriate contexts. The actual mechanism of routing, and a messaging system are described later in Section dist_substrate.

2.1.9 Interoperation

The model implicitly emphasizes the delinking of cause and effect as perceived by a user at the interface. User interactions cause the generation of Action Events, which are routed to the Core *via* the Mapper and the messaging subsystem. This makes the tool amenable to interoperation with other tools. Context messages can be routed to remote tool Contexts to cause actions to be performed. Tools can therefore access remote functionality of any other tool built around a similar model by simply sending the right messages for a request with requisite data, and updating the Context Interface when the response message is received. Tools can block, if desired, while waiting for responses, as in traditional remote procedure calling. Since the only requirement this mechanism imposes is that cooperating tools operate on top of compatible communication mechanisms, this allows for heterogeneous interoperation which transcends implementation language and platform issues. The cooperating units may be instances of the same tool, or even be different tools that operate on the same data.

Different application level protocols can be implemented to support tool-tool interoperation. Support for distribution in the Structural Model is depicted in Figure 2.2. Mechanisms to set up shared Contexts also work in a distributed setting, on top of the messaging subsystem. This is the fundamental feature that enables support for collaboration.

2.2 Media Model

In the previous section we proposed a model that enables us to develop tools amenable to distributed and collaborative operation. In this section we extend the model to build media enabled tools. Once again, the model emphasizes the separation of interface and function, and relies on high level abstraction to enable incorporation of audio, video, graphics, text and domain specific media into tools. In this formulation, any form of structured data that has well defined interaction semantics – documents, spreadsheets, databases, domain specific models, process control data, device control data etc. can be treated as a media type. Multimedia systems have the ability to represent disparate forms of information as a bitstream, enabling a unified storage, processing and communication infrastructure. Here we propose a model for unified sharing and interaction semantics, allowing us to focus on the information rather than on the means of acquiring and presenting it.

The noticeable lack of popularity of multimedia features in current tools, in spite of the vast functionality available for capture and rendition of such information, is primarily due to the lack of an easy way of integrating those facilities into tools. As the use of multimedia become more popular, we will see more tools incorporate multimedia facilities. Our objective is to provide a media-rich substrate for the design of media-enabled tools, by relieving application developers of the burden of low-level device and media manipulation. The model we describe provides a very convenient mechanism to the application developer to incorporate different media facilities into tools without having to deal with any low level issues.

We differentiate the notion of media objects and media streams. A media object is the representation format – raw digital data. Media streams have temporal attributes and denote time based interaction. Thus media objects can be considered discrete, and media streams continuous. Media streams consist of media objects with implicitly or explicitly stored presentation control information. Except for rendition and presentation, the Media Model does not distinguish between the two.

The model is built around the concept of media Agents that are characterized by high-level specifications and a description of their functionality. They provide device independent handling of multiple media types like audio, video, images, 2D and 3D graphics, text, and domain specific models in a heterogeneous setting. Device independence is achieved *via* abstractions that isolate idiosyncrasies of device specific handling and media formats.

Tools incorporate multimedia facilities by inter-operating with these media Agents. Agents are built around the Structural Model. They support the notion of media Sources, Sinks and Filters that provide the mechanism of interacting with the media type. Agents support transport and manipulation of the media object and streams. Sources provide media input. Sinks support media output. Filters are used to apply transformations to the media object or stream after it is input and before it is output.

2.2.1 Agents

An Agent is the actual site of media interaction for a user. It may actually implement media interaction functionality, or may use orthogonal abstract mechanisms to achieve the same effect. Tools are mostly unaware of the existence of the media Agent, The Agent reacts to messages from tools but is otherwise completely transparent to them. It is responsible for managing media real estate, and preventing anarchy in media interaction.

This is especially true in the case of device based media like audio and video. The Agent concept allows for simultaneous use of desktop audio hardware by multiple tools. Similarly, it localizes the issue of camera control for video, allowing it to be used by multiple tools. The same applies for external media device control, like when the computer drives external video and audio playback, recording and transport control hardware, and process control, when the computer drives external machinery or process.

Different media access policies can be implemented in Agents, which implement basic media interaction mechanism. They provide facilities to set up Sources and

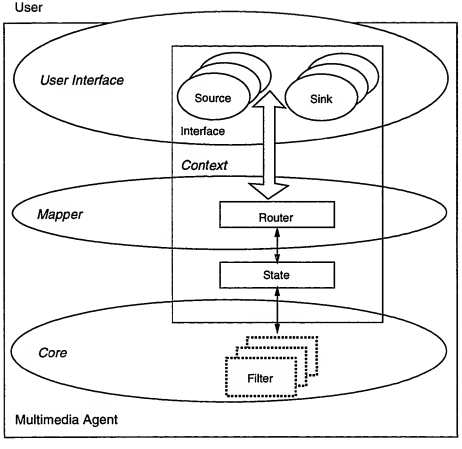

Figure 2.3 Multimedia Application Model

Sinks for the media, and to specify Filters that redirect and transform the media stream. The Structural Model that Agents are built around enables interoperation with other Agents and tools. Agents also provide a substrate with a well defined Application Programming Interface (API) to facilitate developers of tools that need low level media access.

2.2.2 Sources

Media Sources are the starting point of media streams in the address space of the Agent. They may be external processes that actually receive input from media hardware and are capable of communicating with the Agent. They may be local devices, like cameras or microphones, driven directly by the Agent. They may be local streams from secondary storage devices. Or they may be remote streams from other Agents in a distributed setting, brought into the Agent's address space *via* the messaging system and an orthogonal transport process.

2.2.3 Sinks

Media Sinks are the termination point of media streams in the address space of the Agent. As in the case of Sources, they may be external processes that actually send output to media hardware and are capable of communicating with the Agent. They may be local devices, like speakers and video recorders. They may be local streams to secondary storage devices. Or they may be media streams to other Agents in a distributed setting, sent into the remote Agent's address space *via* the messaging system and an orthogonal transport process.

Users perceive media streams at the Sinks. Implementation of Sinks takes into account media specific parameters like its temporal and persistence attributes, presentation and rendition control, etc.

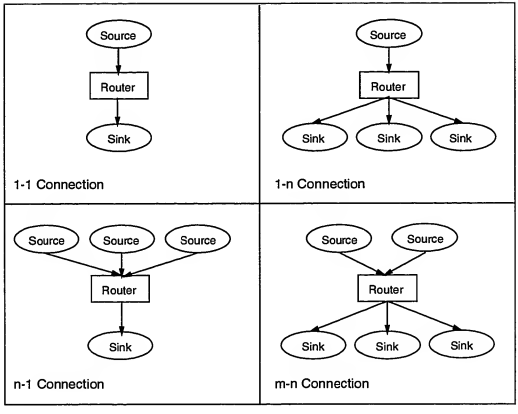

Figure 2.4 Distributed Multimedia Model

2.2.4 Filters

Filters are used to apply transformations to the media stream between Source and Sink. They encapsulate the process of applying these transformations. The fundamental filters provide a mechanism of setting up and tearing down Source to Sink connections. We refer to them as Redirection Filters. They are media independent and enable a single Source to be connected to multiple Sinks, multiple Sources to be connected to a single Sink, and multiple Sources to be connected to multiple Sinks. The semantics of the multiple connections are domain specific. The Distributed Multimedia Model is depicted in Figure 2.4.

Transformation Filters are media specific, and implement functionality like media resampling, format translation, and media processing. *E.g.* in the audio setting they are used to implement amplitude and pitch adjustment, stream mixing, and special effects like echo and reverberation. In the video setting they implement transformation, format translation, brightness and contrast control etc. Since Agents are responsible for managing media real estate, and preventing anarchy in media interaction, they allow simultaneous access to media devices. This process usually employs Filters to enable simultaneous presentation. *E.g.* allowing multiple audio streams to be presented at a sink may involve resampling and mixing if the sampling rates of the streams are different, and differential rate mixing is not supported in the hardware. Many Transformation Filters are mechanisms of user interaction with the media stream.

Filters can also implement media conversion. Speech recognition technology and text to speech technology are now fairly mature. Natural language systems are evolving. Filters implementing these features allow one kind of media stream to be converted and subsequently redirected to Sinks for a different media type.

2.2.5 Interoperation

The Media Model coupled with the Structural Model provides a very convenient mechanism to the application developer to incorporate different media facilities into

applications without having to directly deal with any low level media issues. This is enabled by tool-Agent interoperation. Adding a new media Agent enriches the entire interoperating environment.

2.2.5.1 Media-Unaware Interoperation

In the simplest case the tool is media-unaware, but media-enabled by interoperating with an Agent. It uses the distributed messaging mechanism to request the Agent that deals with the media type to create a Context on its behalf, and sets up the relevant Sources, Sinks and Filters. Actual media interaction thus occurs within the Agent. The Sources and Sinks may exist in the Agent, on stable secondary store or in other tools that this tool is inter-operating with. Their location governs the extent and nature of interaction through the Agent.

In a more complex scenario, tools create media interaction Contexts within their own Contexts. This is usually accomplished using the Agent substrate that provides mechanisms for programmer interaction with the media type, for setting up Contexts, Sources, Sinks and Filters. Agents are dynamically configured to use these client Contexts to render the media data. Media-Unaware Interoperation is based on coarse-grained tool-Agent interaction, and is sufficient for many media-enabled tools.

2.2.5.2 Media-Aware Interoperation

Tools that need to implement low level manipulation of a media type, and require low level control are said to be media-aware. Such tools use the Agent substrate to manipulate the media stream, implement new Source, Sink and Filter mechanisms, or to provide new user interaction methods. This interoperation with the Agent is based on fine-grained interaction with its substrate. This is especially useful when the tool needs low level access, but doesn't need to dynamically interact with the agent, *e.g.* in software only playback of a media stream with a new transport control method.

2.2.6 Media Widgets

Media Widgets are an abstraction mechanism to provide a convenient API to application developers for incorporating media facilities into tools.

For Media-Unaware Interoperation, described above, this is a very lightweight mechanism. Media Widgets in this case are simply stubs that interact with remote Agents that actually implement the functionality. They encapsulate Agent communication and interaction, and provide a well defined API to trigger the interaction, hiding the low level details of actual messaging. In this method the tool instantiates the stub, and uses the API to send messages to the Agent. The tool interacts with the Widget at a very abstract level, in terms of high level media functionality. The stub communicates with the Agent to fulfill requests, and automatically responds to requests from Agents. The application developer is thus shielded from all low-level details.

In Media-Aware Interoperation, tools instantiate a Media Widget. The Widget encapsulates all functionality for the media type. It does not need to communicate with the Agent, except for device control and device based interaction.

The Widgets can implement different levels of functionality. They may support one-shot rendition of the media information, or may support reviewable rendition. Or they may support filtering and manipulation operations supported by Widget-Agent interaction. An example of this scenario is one-time video playback, video playback with transport control, and video playback with image processing capability.

An advantage of separation of Widget and Agent, is that it permits scenarios where a single Agent regulates a device (for example, a video capture board or audio recording hardware). This simplifies contention resolution by centralizing it in the Agent. It provides a mechanism for implementing policy for issues like who can use video or audio hardware, whether or not it can be concurrently used by multiple users, how multiple simultaneous record and playback requests are resolved, and privacy and access issues related to media device usage etc.

The logical and physical separation of Widget and Agent contributes to modularity, since it is essentially a separation of interface and function. Agents can use different hardware and software platforms for implementing the media type, and present only an abstract view to the Widget. Media-Enabled applications are thus made portable across hardware platforms by simply creating an Agent for the new platform.

2.2.7 Heterogeneity

Todays networked environment presents a distributed, heterogeneous hardware setting. Therefore media-enabled tools need to be able to operate on a variety of platforms, in order to be truly useful. The Structural Model emphasizes separation of interface and function, and enables interoperability. The Media Model provides for very high level interaction between tools and Agents. To support heterogeneity, Agents are built on abstract media models that deal with media in a device independent manner. Filters permit the translation to platform specific formats. Thus the Media Model is very amenable to heterogeneity.

2.2.8 Communication

The Media Model assumes the existence of a connection and transport mechanism that underlies the messaging system. Though the implementation issues and inter process communication platform of that layer are orthogonal to the model, the ability to connect multiple Sources to multiple Sinks imposes some requirements on the communication infrastructure. The substrate should provide mechanisms for 1-1, 1-n, n-1 and m-n communication, corresponding to the single Source – single Sink, single Source – multiple Sink, multiple Source – single Sink, and multiple Source – multiple Sink scenarios respectively.

Media properties also dictate certain qualities of the communication substrate. Audio, and especially video, is tolerant to data loss, and can be implemented very efficiently on a datagram based system. However other media types such as models

and spreadsheets need a reliable stream based mechanism for transport. All control information being channeled through the messaging system needs to be passed over reliable mechanisms to prevent inconsistencies. Thus the communication substrate needs to support both reliable and unreliable transport. Available or simulated multicast mechanisms can be employed to implement the n-way transmission requirement.

The Media Model is based on Agents, Sources, Sinks and Filters, as well as portable media streams. All of these units are associated with unique identifiers that serve as their addresses. The distributed messaging system transports routing information to different tools. This allows Source-Filter-Sink connections to be set up and controlled dynamically, and allows media streams to be directed to remote Sinks. This is the fundamental mechanism by which media rich communication occurs in the distributed setting.

We introduce the concept of user gestures, differentiated from user actions in a distributed setting. The notion is domain specific. In general, however, gestural input is usually low level interaction at an interface that causes the tool state machine to make transitions through transient states, and eventually leads to an action. User actions cause transitions between stable states, and are critical units of shared interaction. Gestures, on the other hand, represent interaction that would convey information, but is not critical to the action.

The utility of gestures in a distributed setting may be overshadowed by the cost of transmitting the information. Hence the distinction. Gestures form an important element of interaction, and can be communicated using datagram based non-reliable mechanisms that are used for loss-tolerant media.

2.2.9 Media-Enhanced Interaction

The Media Model coupled with the Structural Model provides a very convenient mechanism to the application developer to incorporate different media facilities into applications without having to directly deal with any low level media issues. Agents

can easily inter-operate with other tools that utilize Agent services to transparently incorporate high level multimedia functionality.

Tool-Agent interoperation can easily be used by mail reader programs built on the Structural Model to present media-rich information to the user, or to capture it from him. It can also be used to build in speech and non-speech audio facilities into tools. The high level of abstraction provided makes it easy to incorporate graphics facilities into tools. In fact, tools can communicate with Agents for different media types which can run simulations, execute database queries, evaluate spreadsheets and perform any sort of domain specific processing before presenting it to the user. Tools achieve extensibility by interoperation, and behave as if Agent functionality is implemented in them.

The Agent Source-Filter-Sink mechanism can be used to implement desktop communication and conferencing facilities. Since Agents transcend simple conferencing and support actual sharing of media objects and streams, they enable maintenance of the notion of a shared information space with reviewable shared material. This can be exploited to support collaborative hypermedia browsing.

An important point to note is that Agents are potentially portable across any platforms because of the high level of abstraction they implement. The Media Model conveniently brings multiple media into the developers realm, and eventually to the users desktop.

Media-enabling facilities of this model are relevant for computer enhanced interaction, computer aided instruction and training, participatory collaborative design, and other cooperative activity involving group processes like discussing, planning, and problem solving.

2.2.10 Multimodal Interfaces

The state transitions in Context state in response to Events can be thought of as the interpretive execution of an embedded command language. In the case of Agents, this takes the form of a media control language. Thus, fundamentally any tool is an

interpreter of an abstract command language. The underlying state machine makes a transition whenever it encounters a complete expression. This abstraction enables us to build tools with multimodal interfaces, where users simultaneously employ multiple input modes to interact with tools. The different methods generate expressions in the command language. Evaluation of these expressions is tantamount to tool execution.

The separation of interface and function enables different interface mechanisms to be plugged in separately or simultaneously to the tool. Agents provide the mechanism to build in multimedia input mechanisms. Thus tools can use speech-to-text conversion filters to have voice or audio cue driven interfaces, and text-to-speech filters for audible interfaces. Alternately, they may have mouse based graphical user interfaces, head and eye tracking based interfaces, pen based interfaces, or touch screen based interfaces etc. Filters for low level event streams from these interfaces can be constructed to produce appropriate expressions in the command language. The different command media can be synergetically used to generate command language expressions from simultaneous multimodal input. Advances in natural language based systems and AI techniques for user interfaces will enable even higher levels of sophistication.

2.3 Collaboration Model

The Structural Model and Media Model, coupled with connection and transport mechanisms, provide a framework for implementing groupware – multi-user tools. These tools are collaboration aware in the sense that they are built around a model that assumes the possibility of simultaneous distributed multi-user interaction. The mechanisms enabled by the described models are policy-free. Here we use them to develop a flexible collaboration model that supports media-enhanced synchronous and asynchronous multi-user interaction.

2.3.1 Tools

The Structural Model implicitly emphasizes the separation of cause and effect in a tool. User interactions cause the generation of Action Events that are routed to the Core *via* the Mapper and the messaging subsystem. This makes the tool amenable to interoperation with other tools. Context messages can be routed to remote tool Contexts to cause actions to be performed. Tools can therefore access remote functionality of any other tool built around a similar model by simply sending the right messages for a request with requisite data, and updating the Context Interface when the response message is received.

The process of creation of a tool can abstractly be thought of as a process of specifying handling mechanisms for Events. Events may be Actions initiated by the user or Triggers originating from some change in tool state or some internal condition. The tool behaves as a state machine that reacts to Events by making state transitions, possibly generating synthetic Events in the process. If we assume the immutability of tool state other than by this process, we have a model for collaborative tools. In this model, a shared Context receives Events simultaneously from multiple tools *via* the messaging system. These Events cause a state change in the shared Context. This activates Triggers that cause messages to be sent to other instances of the shared Context.

A distributed tool provides output to multiple remote Contexts and does not accept or handle remote input. It just provides a mechanism of displaying state in a distributed setting. A collaborative tool handles local and remote input. It provides the mechanism for synchronous and asynchronous collaborative interaction. Tools create and manage Contexts on behalf of other tools that they are linked to. This process of Proxy Context management is the fundamental mechanism for building distributed and collaborative tools.

In the case of a local Context, and for output-only remote Contexts, only one thread of control modifies state of the Context. In the collaborative case multiple

threads of control, one local and multiple remote, modify Context state. This necessitates handling of issues of atomicity of operations and concurrency control to prevent inconsistencies. Separation of user interface and Context state, which allows state to be altered by mechanisms other than the user interface, is crucial.

Tight coupling of state and interface is desirable, so that the interface always depicts current state. In this model, there is little difference between collaborative and non-collaborative use of tools. All Events identify the Context they occur in and the Contexts they affect, and alter State of those Contexts. This change is subsequently reflected at the Interface. Delinking of cause and effect makes distributed interoperation transparent, barring performance considerations in the synchronous case.

2.3.2 Consistency

Collaborative interoperation centers around the maintenance of mutual consistency. All shared Contexts need to have identical state, though States may be partially mutated temporarily for performance reasons.

Interaction of tools with external state, like files on a local disk, must explicitly be captured and introduced into the shared State, if shared interaction over that information is required. Multi-user tools can therefore be guaranteed input consistency, output consistency, and startup consistency for shared State.

The separation of Context State and Interface allows for different views of the same state, and even allows different kinds of interaction simultaneously. This allows for implementation of different degrees of coupling between Interfaces, and of undo-redo models for flexible shared interaction.

2.3.3 Collaboration

The Structural Model and Media Model allow us to implement different Collaboration Models by setting up shared remote Contexts between tools. These tools are

collaboration aware in the sense that they are built around a model that assumes the possibility of simultaneous distributed multi-user interaction.

We discuss how we can more efficiently and easily implement the two traditional approaches and also propose a new approach. The simplest is the Centralized model, where one instance of a tool maintains multiple interfaces in a distributed setting, and is thus shared. In the Replicated model, multiple instances of tools cooperatively maintain shared state and interfaces. The Session model incorporates the desirable features of the two traditional models, and enables persistence.

2.3.3.1 Centralized Model

The simplest collaboration model is the Centralized model. In this model one tool in the cooperative environment maintains multiple interfaces in a distributed setting. User interaction may occur at any interface, and the tool and its state is thus shared.

The Centralized model is implemented in the Collaboration Model by setting up a Context in a tool, and sharing it with remote Contexts. Events from all interfaces are routed to the central Context, and state change information is relayed to all the remote Contexts, generating Triggers that generate updated views at the Interfaces. Collaborative tasks are implemented in the shared Context and State of the single central tool. Simultaneous interaction is supported from multiple distributed interfaces.

Screen and window sharing is one traditional mechanism of implementing centralized collaborative tools. Here, low level mechanisms are employed to intercept interface display commands from a collaboration unaware tool. The streams are relayed to multiple distributed sites where the interfaces are recreated. Events from all the interfaces are directed to the central application. The interception mechanism implements and enforces a turn taking policy to prevent inconsistency. A major drawback of the screen sharing approach is that existing systems are built on the greatest common denominator platform, and cannot take advantage of available local facilities in a heterogeneous setting. View generation for all interfaces is done in the central

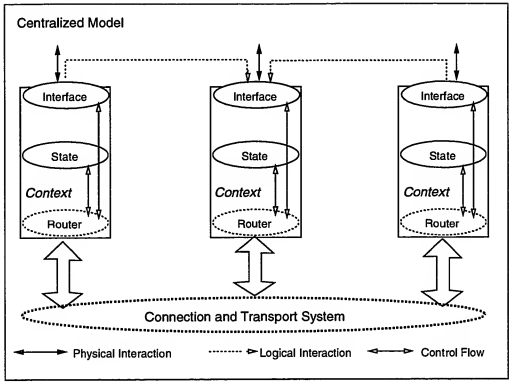

Figure 2.5 Centralized Collaboration Model

site. This is a major bottleneck in the distributed setting, and does not scale well as number of interfaces managed is increased. More intelligent interception mechanisms can be built, in theory. In practice, however, they would often need to understand the application domain to be able to translate interface streams.

Another traditional centralized model is based on collaboration aware tools. Here, one instance of a tool internally creates and manages multiple distributed interfaces. Such tools implement a finer degree of sharing than the earlier approach, by handling multi-party interaction. Centralized view generation is still a bottleneck. Also, every element of interaction involves transmission over the network, and computation in the central tool. This doesn't scale well as the number of managed interfaces is increased, since there is a performance penalty for every added interface. This is especially true

if the tool operates in a heterogeneous environment and uses different techniques to generate views on different platforms. Centralized systems thus implement the Single-Input Replicated-Output or the Serialized-Input Replicated-Output paradigm.

Our model offers a better implementation of the collaboration aware centralized single application model. Even though functionality is centralized in the Core of the central tool, the Interfaces of shared Contexts are intelligent and maintain enough state information to be efficient in a distributed setting. The implementation is depicted in Figure 2.5.

The decentralization of interface and view generation makes it much more efficient in a distributed setting by minimizing the amount of communication between collaboration Contexts. This reduces the performance penalty involved in adding more interfaces. An additional advantage is that Interfaces can present different views of the same State, and can offer different interaction mechanisms. The model provides mechanisms for enforcement of different floor control policies, and supports simultaneous interaction streams that are eventually serialized in the central tool Core. The biggest advantage is that such systems are easy to build because the central tool only has to set up new shared Contexts when they are added, and tear them down when they are removed. The messaging system automatically channels Event streams, and handles collaborative interaction. Tools can be involved in multiple independent collaborations using independent Contexts.

2.3.3.2 Replicated Model

Another collaboration model is the Replicated model. In this model multiple collaboration-aware tools in the cooperative environment cooperatively maintain the notion of shared state and interaction at multiple interfaces in a distributed setting. User interaction may occur at any interface to affect the collaborative state. Collaborative tools based on the Replicated model have the advantage that they support cooperative manipulation of shared state, and thus support simultaneous multi-party interaction in the true sense. They can be set up to provide independent views of

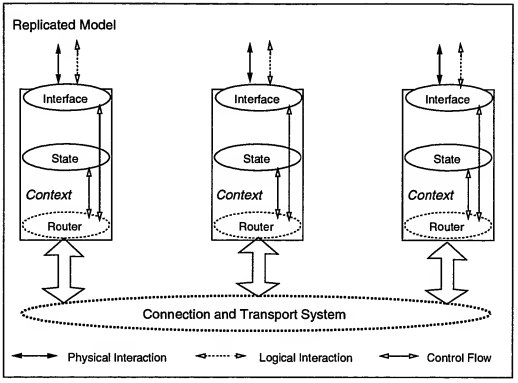

Figure 2.6 Replicated Collaboration Model

shared state. They are potentially more amenable to platform heterogeneity. Tools based on this model are built around replicated data management facilities.

A drawback of traditional replicated systems is that they operate on the premise of shared input. Computationally expensive tasks are performed at every participating site, losing the benefit of sharing from distribution. Also, for highly inter-related tasks and the associated data updates, replicated data management facilities do not scale very well, since all data sites have to be kept synchronized. They outperform centralized systems when tasks and data updates are unrelated.

The Replicated model is implemented in our Collaboration Model by setting up connected shared Contexts in multiple tools. In one method, Events from all interfaces are routed to all Contexts. Context State changes generate Triggers that update views

at the Interfaces. Tools explicitly implement replicated data management facilities. In another method, Contexts handle user Actions locally, and use an underlying replicated data management system to update shared State. State changes generate Triggers that update views at the Interfaces. In both methods, collaborative tasks are implemented in the shared Context and shared State of all tools. Simultaneous interaction is supported from multiple distributed interfaces. The implementation is depicted in Figure 2.6.

One advantage of our model is that it allows both input and output replication. Input replication for low computation tasks allows us to exploit the parallelism that stems from distribution. Output replication for compute intensive tasks allows us to take advantage of the sharing that distribution enables. The biggest advantage of our model is that such systems are easy to build because the tools only have to set up new shared Contexts when they are added, and tear them down when they are removed (The notions of Adding and Removing group members). The messaging system automatically channels Event streams, and handles collaborative interaction. Tools can be involved in multiple independent collaborations using independent Contexts.

2.3.3.3 Session Model

In this model, a Session is the unit of collaborative activity. A Session is essentially a Context without an Interface. Session Model based collaborative tools are implemented in our Collaboration Model by instantiating a Session, that causes the setting up of connected shared Contexts in multiple tools. These shared Contexts are collaborative task aware. Events that are associated with low computation tasks are routed to the Session Context, which relays them to all shared Contexts. Events that are associated with compute-intensive tasks are acted upon in the tool Context, and the associated Triggers are routed to the Session Context. Context State changes generate Triggers that are routed to tools and update views at their Interfaces. The implementation is depicted in Figure 2.7.

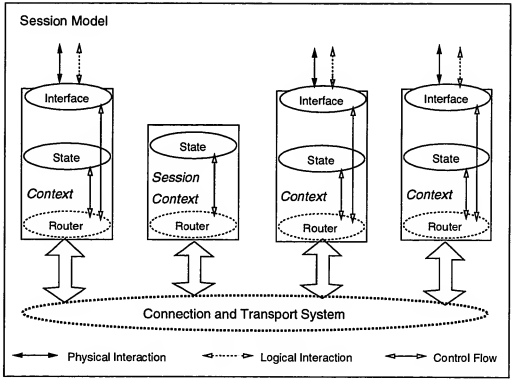

Figure 2.7 Session Model of Collaboration

Since Sessions are collaborative task-aware, they can choose between centralized and replicated data management facilities based on the number of sites in the collaboration, degree of dependence between collaborative tasks, and performance of the underlying mechanisms.

Collaborative tasks are thus implemented in the shared Context and State of a Session. Simultaneous interaction is supported from multiple distributed interfaces.

A major advantage of this approach is that Sessions can be made persistent, since they are delinked from user level tools and interfaces. They can be saved and restarted, and thus support asynchronous and synchronous collaborative interaction. Also, participating in collaborative tasks is further simplified, since tools do not have to keep track of group membership, or set up routing information. Tools create Contexts that are shared with the Session Context when they join a Session, and tear them down when they leave.

2.4 Meeting CSCW Requirements

We have developed a model for groupware, based on models for tool structure and media integration. The model enables convenient development of synchronous and asynchronous multi-user tools, based on different collaboration models. The proposed model is abstract, and makes no assumption about implementation language and platform of groupware, and imposes no restrictions. The only requirement it has is that the underlying messaging system allow for communication in a distributed setting. It makes certain recommendations about the underlying transport mechanism in the interest of runtime performance.

In this section we revisit the requirements of a CSCW enabling infrastructure, and discuss how the model fulfills or supports the requirements.

2.4.1 Shared Data Management

The Collaboration Model is flexible in terms of the actual mechanisms for transporting data between shared Contexts in a collaboration. Centralized or replicated

data management facilities may be part of the underlying messaging system, or can be implemented on top of it. Alternately, mechanisms can be an independent subsystems like distributed databases, or distributed object sharing systems. Shared data management mechanisms need to support the ability to trigger changes in views when data is changed.

The notion of multiple Contexts in tools, some private and some shared, allows for easy support of separation of private and shared workspaces. Tools can provide mechanisms to move data from private to shared workspaces, and vice versa.

2.4.2 Distribution Control

The Structural Model emphasizes the separation of cause and effect, that enables distributed interoperation. Actions can be routed to remote Contexts, and data transfer activates Triggers that update the interface. The Collaboration Model is orthogonal to connection and transport mechanisms, and can use existing technologies. Mechanisms that offer both datagram (unreliable) and stream (reliable) communication, and support 1-1, 1-n, n-1 and m-n way communication are recommended for efficient implementation.

2.4.3 Concurrency Control

If shared data management is implemented on top of shared independent systems like distributed databases, concurrency control is automatically taken care of, since those technologies implement it internally. The separation of cause and effect, that of interface and function, enables the implementation of any of the well known concurrency control mechanisms. Concurrency control is simpler if data management is centralized.

2.4.4 Session Control

The model allows for flexible collaboration control mechanisms that regulate how multiple users assemble and interact over shared data. The Structural Model allows

for interoperation with agents in the environment that keep track of user location and activity in the distributed setting. This interoperation can be exploited to regulate session setup and tear down, formation of collaborative groups, and dynamic inclusion and removal of participants. Flexible collaboration control methods to initiate and terminate collaborative sessions, to join or leave ongoing sessions, and to invite participation in collaborative tasks can thus be implemented.

2.4.5 Interaction Control

The Collaboration Model allows for synchronous multi-party interaction. However, interoperation can be used to implement floor control and interaction regulation policies for users by dynamically controlling which Interfaces are allowed to affect shared Contexts. This can be exploited to efficiently implement flexible scenarios where specific numbers of individuals interact in shared Contexts. Different flexible and intuitive protocols for requesting, taking and giving up turns can be implemented.

2.4.6 Coordination Control

The Collaboration Model allows for synchronous multi-party interaction, where everyone is allowed to do everything in a shared context. Shared Context Interfaces present continually updated views of collaborative activity, and provide awareness. Flexible coupling policies can be implemented that control the granularity of transmission of awareness information. Facilities that dynamically control rates at which Gestures and other communication elements are transmitted, can be provided to both tools and users.

Tool interoperation can be used to implement access regulation policies for users by dynamically controlling which Events are allowed to affect shared Contexts. Further, if all objects in shared Contexts are assigned unique identifiers, mechanisms that regulate finer grained access can be implemented. This can be used to convey notions of object ownership, and to regulate what actions different users can perform on

different objects and their parts. This can also be used to specify divisions of labor to coordinate collaborative activity.

2.4.7 Multimedia and Graphics

The Structural and Media Models enable painless integration of multiple media facilities into applications, enabling media rich communication. Media is treated as structured data with specific interaction semantics, and is therefore subject to Interaction Control and Coordination Control, allowing for flexible interaction over shared media.

2.4.8 Collaborative User Interfaces

The Structural, Media and Collaboration Models provide a mechanism for building sophisticated multi-user interfaces for collaborative tools. A collaborative tool can be thought of as a distributed data-flow machine. Interoperation provides a mechanism for dynamically controlling the behavior of the virtual machine. Media integration in conjunction with novel input and output mechanisms can be exploited to build collaborative environments for problem solving.

3. SYSTEM ARCHITECTURE

3.1 Introduction

Recent strides in electronics, computer and communication technology have resulted in proliferation of multimedia workstations. This has provided us with extremely powerful tools on the desktop. The need for architectures and abstractions to build computer mediated cooperation mechanisms is heightened by our rapid progress towards the information revolution that will be ushered in by facilities built on top of these desktop machines. Computer supported cooperative interaction, incorporating information exchange and multimedia communication will revolutionize how we collaborate to solve problems and how we work.

In Section 1 we introduced CSCW, and described the requirements on an enabling infrastructure. In Section 2 we proposed models for tool structure, media integration and collaborative interoperation that support high levels of abstraction to aid in groupware development. In this section we describe a prototype for an enabling CSCW infrastructure, based on those models, that targets and fulfills all the requirements.

The proposed models enable us to transcend implementation details. Tools can be built using any language and will interoperate as long as they are built around the Structural Model, and use a compatible messaging and communication mechanism. We use the pervasive implementation environment of C, Unix, X11 [102, 101], and the networking platform of TCP/IP [33] to express the substrate. We create multiple layers of abstraction with well defined interfaces. This provides us the flexibility of switching to more advanced mechanisms as they evolve.

In the Structural Model we described a cooperative tool paradigm that defined tool architectures amenable to building conferenced tools. This approach entails

integrating a collection of function-specific tools into a distributed and extensible environment. In this setup a tool can easily use facilities provided by other tools using remote procedure calling. This tool-level cooperation allows us to exploit the commonality that is inherent to related tools. An infrastructure of communication and interaction tools, display and visualization facilities, symbolic processing substrates, and simulation and animation tools saves avoidable re-implementation of existing functionality, and speeds up the application development process.

3.1.1 Requirements

A communication substrate needs to provide facilities for connection setup in a distributed setting, as well as mechanisms for transport of data between multiple hardware platforms in a heterogeneous network. In order to empower media-rich communication, the substrate needs to support text, audio, video, 2D graphics and 3D graphics messaging, in addition to application-specific models and data. It needs to facilitate synchronous and asynchronous exchange of multimedia information. Such information is useful to successfully communicate at the time of design, and to share the results of tasks, and is often necessary to actually solve problems.

A collaboration substrate needs to facilitate startup of conferences – multi-user collaborative sessions, between users linked by a network. This entails mechanisms for initiation of sessions, invitation to sessions, and assimilation into sessions. The infrastructure must support conduction of such collaborative sessions *via* notions of collaborating groups and shared contexts. It must perform access regulation and should be dynamically configurable to support different kinds of multi-user interaction modes, so that users can cooperate in turn-taking based scenarios or synchronous multi-point conferences.

Importantly, the CSCW infrastructure must provide a convenient abstraction to the application developer, shielding him from lower level details, while providing him with a rich substrate of high level mechanisms. This would make it easy to design multi-user tools, effectively harnessing current technology to build powerful

collaborative virtual machines. At the same time, the CSCW infrastructure should enable a convenient, non-intrusive environment to motivate end users to cooperate in their problem solving efforts.

3.1.2 Features

Shastra is an extensible, distributed and collaborative geometric design and scientific manipulation environment. It consists of a static and a dynamic component. The static component is a CSCW infrastructure for building scientific CSCW tools. We call it the Shastra Layer. It defines an architectural paradigm that specifies guidelines on how to construct tools that are amenable to interoperation. Its connection and distribution substrate facilitates inter-tool cooperation. Its communication substrate supports data sharing and transport of multimedia information. Together they promote distributed problem solving for concurrent engineering. The collaboration substrate supports building synchronous multi-user tools by providing session management and interaction control and access regulation facilities.

In addition to the distribution, communication and collaboration framework, Shastra provides a powerful numeric, symbolic and graphics and multimedia substrate. It enables rapid prototyping and development of collaborative software tools for the creation, manipulation and visualization of multi-dimensional geometric data.

The dynamic component of Shastra is a runtime environment that exploits the benefits of the architectural philosophy and provides runtime support for conferenced tools. It is described in Section 4.1.

The CSCW infrastructure of the Shastra system facilitates creation of collaborative multimedia tools. We adopt an abstract tool architecture that enables inter-tool communication and cooperation. It supports remote task invocation and brokering. We propose a hybrid computation model for CSCW tools that is very effective in a heterogeneous environment. The system provides intuitive session initiation methods, flexible interaction modes, and dynamic access regulation.

3.1.3 Two Level Enabling

The design of Shastra is the embodiment of a simple idea – software tools can abstractly be thought of as objects that provide specific functionality. These objects exchange messages, automatically or under user command, to request other objects to perform operations. The CSCW infrastructure of Shastra specifies architectural guidelines and provides communication facilities that let tools cooperate and exchange information to utilize the functionality they offer. This enables cooperation at the tool level. The infrastructure provides collaboration and multimedia facilities allowing the development of tools in which users collaborate to solve problems. This enables user level cooperation. A synergistic union of these two ideas lets us design sophisticated problem solving virtual machines.

3.2 Architecture

Tools in Shastra are built with the underlying idea of inter-tool cooperation. Every tool is abstractly composed of three layers. The Core is accessed through any of the Interfaces *via* a Mapper. The application-specific Core implements the functionality offered by the tool. Above the Core is a functional Interface Mapper that invokes functionality embedded in the Core in response to requests from the Graphical User Interface, ASCII Interface or the Network Interface. It also maps requests to alter the user interface or to send messages on the Network Interface. The Mapper is essentially a command interpreter that invokes registered event handlers when events of interest occur. Tools register event handlers with the Mapper for events they are interested in, and unregister those that cease to be of interest.

The separation of Core and Interface, that of function and interface, makes it easy to build multi-user systems, since it enables the maintenance and display of shared state at a user interface *via* remote commands in a distributed system.

The GUI is application-specific. The ASCII interface is a shell-like front end for the tool. Tools communicate with other tools in the environment, *via* the Shastra

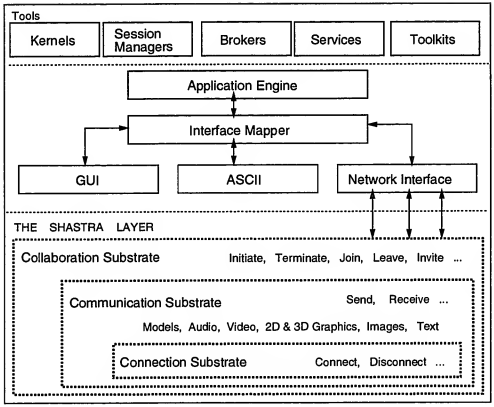

Figure 3.1 High Level Architecture of a Tool in the Shastra Environment

substrate, through an abstract Network Interface. This implements the underlying messaging system that provides connection and transport facilities. The Network Interface multiplexes multiple simultaneous network connections, and implements the different application level communication protocols [8]. Functionality available at a network interface is described to the communication substrate using a signature that specifies callback functions for the different kinds of network events that the tool is interested in. The signature provides an abstract interface to remote systems, and describes functionality offered by the tool. It also serves as a regulatory mechanism, since different levels of service can be offered at different interfaces by specifying the appropriate signatures.

To take advantage of the integration facilities of the infrastructure, the Core uses the Network Interface to access functionality already implemented in other tools. The main benefit from this setup is modularity and reuse – tools isolate the functionality they offer, and provide a functional interface to peers. The high level block architecture of tools in Shastra is depicted in Figure 3.1. The architecture makes it easy for tools to connect to other tools and request operations, synchronously as well as asynchronously.

These architectural guidelines accord us the benefit of uniformity since all tools are built upon a common infrastructure and have identical connection, communication and collaboration mechanisms. The concept of cooperation awareness thus pervades the architecture. The entire set of connected Network Interfaces of Shastra tools manifests itself as the abstract Shastra layer at runtime (see Figure 3.1). It maintains the collaborative environment, provides access to functionality of different systems, and provides facilities for initiating, terminating, joining, leaving and conducting collaborations. The connected network interfaces of Shastra tools comprise a distributed virtual machine on which we build problem solving applications.

The enabling substrates use the event paradigm to provide functionality. Tools use the application programming interface of the substrate to cause request messages to be sent over connections. Tools interested in any event register handler functions for

it with the Mapper. The handler functions are invoked when that event is received. This allows tools to take action appropriate to the event when it occurs.

3.2.1 Distribution Substrate

It provides mechanisms of setting up and tearing down tool-to-tool connections in a distributed setting. It provides device independent data structure transport for heterogeneous environments. It implements synchronous and asynchronous remote procedure calling and provides multiple-connection management between instances of tools. It supports several application level communication protocols. Coupled with the data communication facility, this enables flexible management of tool state.

Tools interact with the connection subsystem using the following messages and events.

- Connect_Request – Tools send this message to initiate a connection with a remote tool. Arguments specify the destination, whether a reliable or unreliable connection is desired, and the protocol to be used for that connection.

  At the other end, the connection subsystem sends this event to the Mapper when a remote tool attempts to connect to the local tool. The event handler function allows a tool to control whether or not to accept a connection request, and to set up the local side for remote interaction if the request is accepted. A Connect_Notify message is then usually sent by the tool, with the appropriate information.

- Connect_Notify – The connection subsystem sends this event to the Mapper when a connection is established from the local tool to a remote tool, or when the connection is refused, typically in response to a Connect_Request. The event handler function allows a tool to set up the local side for remote interaction.

- Disconnect_Request – Tools send this message to terminate an existing connection with a remote tool. Arguments identify the connection to be torn down.

The connection subsystem gracefully terminates the connection, and frees associated data structures.

- Disconnect_Notify – The connection subsystem sends this event to the Mapper when an established connection is terminated. The event handler for this event allows a tool to clean up the local side at the end of remote interaction.

- QueryState_Request – Tools send this message to the communication subsystem to query the state of a connection, that is identified by the argument.

The interface of this substrate specifies only mechanism. The current implementation uses Unix sockets to establish connections across the network. Stream, datagram, and multicast connections are currently supported, implemented on top of the TCP, UDP and UDP Multicast. Application level protocols with and without acknowledgement, and with and without sequencing are implemented.

Tools interact with the transport subsystem using the following messages and events.

- Send_Request – Tools use this message to cause the transport subsystem to transmit data.

  At the other end, the transport subsystem sends this event to the Mapper when data is received on the connection. The event handler function lets tools take appropriate action when data is received.

- Send_Notify – This message is used only if the application level protocol uses acknowledgement. The receiving site sends this message on receipt of data.

  The transport subsystem of the data transmitting site sends this event to the Mapper when data is successfully sent on an established connection. The registered event handler enables the tool to take appropriate action.

- Receive_Request – Tools use this message to synchronously wait for data on an established connection.

At the remote end, the transport subsystem sends this event to the Mapper. The event handler for this event usually responds with a Send_Request, to send data to the waiting tool.

- Receive_Notify – The transport subsystem sends this event to the Mapper, typically when it receives response data from a Receive_Request. This indicates delivery of requested data.

The distribution substrate operates in a heterogeneous setting by using XDR (External Device Representation) encoding and decoding of Sun RPC for maintaining platform independent data structure representations.

By default, data transport in an event based system is asynchronous. The transmitter writes to the virtual pipe. When the entire packet is received at the receiver's end, a Send_Request event is generated, and the data is handled. If the protocol uses acknowledgement, the receiver sends an acknowledgement, and a Send_Notify event is generated in the sender. Connections can be operated synchronously by blocking on responses by invoking registered event handlers immediately after sending request messages. In this mode, remote interaction is completely transparent to the tool, except for performance considerations.

The distribution substrate provides its functionality *via* the abstraction of Distribution Widgets. These widgets encapsulate the underlying messaging, and transparently provide distributed functionality like remote procedure calling. Distributed features are incorporated in tools by creating and manipulating these widgets. They support the notion of callback functions to allow tools to take actions when different events occur.

3.2.2 Collaboration Substrate

It uses the distribution toolkit to implement distributed shared state and context. It provides a session management, interaction control, and access regulation facilities. It provides the mechanism to implement policies governing these issues. It enables

rapid prototyping and development of collaborative tools and groupware. The main features are described in [11, 9].

Tools interact with the session control subsystem using the following session control messages and events.

- Initiate_Request – This message is used by a tool to create a new collaborative session. A session is the unit of collaborative activity, and encapsulates shared state and interaction. Arguments specify initial properties of the session.

- Initiate_Notify – The session control subsystem generates this event after a new session is created by a tool. The event handler, invoked when this event is processed, provides tools with a mechanism to take action when a collaborative session starts. It usually involves the creation of a shared Context for the collaborative activity.

- Join_Request – A tool uses this message to request admittance to an ongoing collaborative session. The session is identified using arguments.

  At the receiving end, the handler for this event enables tools to implement different policies and user control over session participation. It sends a Join_Notify message which indicates whether or not the requesting tool is allowed to join the ongoing session.

- Join_Notify – The session subsystem sends this event to the Mapper, typically when it receives a Join_Notify message from a remote tool. The handler function allows the tool to set up a shared Context if it was admitted to the session.

- Leave_Request – Tools use this message to leave a collaborative session.

  At the receiving end, the session subsystem sends this event to the Mapper. The handler function sends a Leave_Notify message, and deletes data structures associated with the shared Context specific to the leaving tool.

- Leave_Notify – The session subsystem sends this event to the Mapper, typically when it receives this message in response to a Leave_Request. The handler

function allows the tool to delete the shared Context associated with the session, and free associated data structures.

- Invite_Request – Tools send this message to invite other tools to join an ongoing session that the inviter is a part of. The session is identified using arguments.

  At the receiving end, the session subsystem sends this event to the Mapper. The handler function allows tools to implement session joining policies, and user interface mechanisms to prompt the user for invitation. The handler function sends an Invite_Notify with the user response.

- Invite_Notify – The session subsystem sends this event to the Mapper, typically when it receives this message in response to an Invite_Request. The handler function usually sends a positive Join_Notify message to the responding tool if it accepted to join.

- Remove_Request – Tools send this message to request removal of other tools from the session. Target tools are specified as arguments.

  At the receiving end, the session subsystem sends this event to the Mapper. The handler function implements policies for removal of tools from sessions. A Remove_Notify message is sent to the sender, and the target tool is removed if the implemented policy allows it.

- Remove_Notify – The session subsystem sends this event to the Mapper, typically when it receives this message in response to a Remove_Request. The handler function allows tools to take appropriate action.

- Terminate_Request – Tools use this message to request the stopping of a collaborative session.

  At the receiving end, the session subsystem sends this event to the Mapper. The handler function implements session termination policies. A Terminate_Notify message is sent to the sender, with the appropriate information about whether or not the session was allowed to terminate. If allowed, the session terminates.

- Terminate_Notify – The session subsystem sends this event to the Mapper, typically when it receives this message in response to a Terminate_Request. The handler function allows tools to take appropriate action.

- Suspend_Request – Tools use this message to request a session to save its state on stable store for future restoration.

  At the receiving end, the session subsystem sends this event to the Mapper. The handler function implements session suspension policies. A Suspend_Notify message is sent to the sender, with the appropriate information about whether or not the session was allowed to suspend. If allowed, the session saves its state to stable store.

- Suspend_Notify – The session subsystem sends this event to the Mapper, typically when it receives this message in response to a Suspend_Request. The handler function allows tools to take appropriate action.

- Restore_Request – Tools use this message to restore a session that was suspended, by instantiating a new session and overlaying its state with that from stable store. The saved file from which restoration occurs is specified *via* arguments.

  At the receiving end, the session subsystem sends this event to the Mapper. The handler function implements session restoration policies. A Restore_Notify message is sent to the sender, with the appropriate information about whether or not the session was allowed to restore. If allowed, a session is instantiated, and its state is restored from stable store.

- Restore_Notify – The session subsystem sends this event to the Mapper, typically when it receives this message in response to a Restore_Request. The handler function allows tools to take appropriate action.

Tools interact with the interaction control subsystem using the following interaction control messages and events.

- Format_Request – Tools use this message to request specification of the format of a session. The requested format is specified as an argument. Session format is a policy that governs how users (*via* other tools) are inducted into sessions. *E.g.* sessions may be open to all users or may be based on invitations alone.

  At the receiving end, the session subsystem sends this event to the Mapper. The handler function implements the desired session format policies. A Format_Notify message is sent to the sender, with the appropriate information.

- Format_Notify – The session subsystem sends this event to the Mapper, typically when it receives this message in response to a Format_Request. The handler function allows tools to take actions appropriate to the set format. The session format is provided.

- Mode_Request – Tools use this message to request specification of the interaction mode of a session. The requested mode is specified as an argument. Session mode governs issues like whether all participants can interact synchronously over the shared Context, or whether they need to take turns to interact.

  At the receiving end, the session subsystem sends this event to the Mapper. The handler function implements the desired session mode policies. A Mode_Notify message is sent to the sender, with the appropriate information.

- Mode_Notify – The session subsystem sends this event to the Mapper, typically when it receives this message in response to a Mode_Request. The set mode is provided. The handler function allows tools to take actions appropriate to the set mode.

- Floor_Request – Tools use this message to request granting of the floor of the session. Floor essentially refers to a "turn" to interact.

  At the receiving end, the session subsystem sends this event to the Mapper. The handler function implements the desired floor request handling policies. A

Floor_Notify message is sent to the sender, with appropriate information about whether or not it was granted the floor.

- Floor_Notify – The session subsystem sends this event to the Mapper, typically when it receives this message in response to a Floor_Request. It specifies the tool which gets granted the floor. The handler function allows tools to take actions appropriate to whether or not they have the floor.

- FloorControl_Request – Tools use this message to request specification of the floor control policy of the session. Floor control policy governs issues of how floor is relinquished and assigned, whether or not it is preemptable, etc.

  At the receiving end, the session subsystem sends this event to the Mapper. The handler function implements the desired floor control policies. A FloorControl_Notify message is sent to the sender, with information about the current floor control policy.

- FloorControl_Notify – The session subsystem sends this event to the Mapper, typically when it receives this message in response to a FloorControl_Request. It specifies the floor control policy. The handler function allows tools to take actions appropriate to the set policy.

Tools interact with the access regulation subsystem using the following access regulation control messages and events.

- SetCapability_Request – Tools use this message to request setting of capabilities of a tool in the session. The target tool and the capability specification are specified as arguments. Capability governs issues of what actions users (tools) are allowed to perform in the session context – whether they can modify session state etc.

At the receiving end, the session subsystem sends this event to the Mapper. The handler function implements the capability control policies. A SetCapability_Notify message is sent to the sender, with information about the current capability specification of the specified tool.

- SetCapability_Notify – The session subsystem sends this event to the Mapper, typically when it receives this message in response to a SetCapability_Request. It specifies the target tool and the capability specification of that tool. The handler function allows tools to take appropriate action.

- GetCapability_Request – Tools use this message to request querying of the capabilities of a tool. The target tool is specified as an argument.

  At the receiving end, the session subsystem sends this event to the Mapper. The handler function returns the capability specification of the specified tool. A GetCapability_Notify message is sent to the sender, with information about the current capability specification of the specified tool.

- GetCapability_Notify – The session subsystem sends this event to the Mapper, typically when it receives this message in response to a GetCapability_Request. It specifies the target tool and the capability specification of that tool. The handler function allows tools to take appropriate action.

- SetPermissions_Request – Tools use this message to request setting of permissions for shared objects in the session. The target object and the permissions specification are specified as arguments. Permissions govern issues of what actions users (tools) are allowed to perform on session objects – whether they can modify those objects etc.

  At the receiving end, the session subsystem sends this event to the Mapper. The handler function implements the permissions control policies. A SetPermissions_Notify message is sent to the sender, with information about the current permissions specification of the specified object.

- SetPermissions_Notify – The session subsystem sends this event to the Mapper, typically when it receives this message in response to a SetPermissions_Request. It specifies the target object and the permissions specification of that object. The handler function allows tools to take appropriate action.

- GetPermissions_Request – Tools use this message to request querying of the permissions of a shared object. The target object is specified as an argument.

  At the receiving end, the session subsystem sends this event to the Mapper. The handler function returns the capability specification of the specified tool. A GetPermissions_Notify message is sent to the sender, with information about the current permissions specification of the specified object.

- GetPermissions_Notify – The session subsystem sends this event to the Mapper, typically when it receives this message in response to a GetPermissions_Request. It specifies the target object and the permissions specification of that object. The handler function allows tools to take appropriate action.

The collaboration substrate provides its functionality *via* the abstraction of Collaboration Widgets. These widgets encapsulate the underlying messaging, and transparently provide collaborative functionality like session control, interaction control, and access regulation. Collaborative features are incorporated in tools by creating and manipulating these widgets. They support the notion of callback functions to allow tools to take actions when different events occur.

3.2.3 Portable Graphics

The XS Hardware-Independent Graphics System is described in detail in [12]. XS is a powerful mechanism for engineering 3D graphics based user interfaces for software tools. It consists of a suite of libraries that provide access to system-dependent graphics facilities in a uniform, system-independent manner. This makes these tools portable to different hardware or software graphics platforms. Each graphics system supported in XS is represented by a library in the suite. All libraries implement the

same graphics paradigm and present the same application programming interface, permitting maintenance of source-level portability across several systems in application programs.

The current XS suite includes libraries for

- SGI workstations which implement the GL graphics library.
- HP workstations which implement the Starbase graphics library.
- X11 based workstations which implement Xlib, the X graphics library
- IBM-compatible personal computers which implement Windows graphics library.

Figure 3.2 shows the block architecture of XS and the platforms supported. XS provides facilities for graphics window manipulation, viewing and modeling control, as well as facilities for drawing 3D objects *via* primitives like points, lines and polygons. It also provides control over color and material properties of graphical objects, scene illumination, and shading and texturing control. The vector and matrix manipulation substrate is used to implement a powerful graphics engine, supporting the notion of nested coordinate spaces.

XS implements graphics features in software if they are not available in hardware. *E.g.* the X11/Xlib version of XS is a purely software implementation. Adding libraries for new hardware platforms to XS automatically supports XS based tools on that architecture.

XS enables application developers to maintain and manipulate a high level abstraction for 3D graphics, and enables rapid prototyping of portable graphics tools. Graphics Agents (The notion of Agents was introduced in Section 2.2) in the Shastra environment are built on top of XS. They enable inclusion of 3D graphics facilities into tools, based on very high level interaction.

Universal adoption of standards like PEX, PHIGS and OpenGL would eliminate the need for a system like XS. However, it seems unlikely that any one of those systems

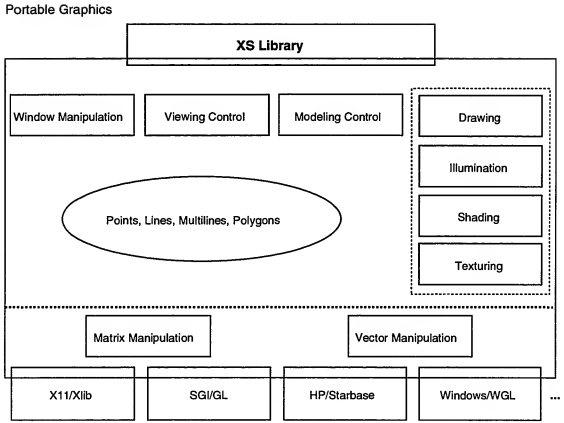

Figure 3.2 High Level Architecture of XS

will prevail, since all of them have large and intransigent following. This makes XS truly valuable as a portable graphics engine.

3.2.4 Collaborative Graphics Substrate

It is based on the structural model and uses the distribution substrate and the graphics substrate to implement device independent distributed and collaborative graphics. It supports synchronous and asynchronous 2D and 3D graphical interaction in a heterogeneous setting. It provides high level control of display and visualization parameters and telepointing.

Tools interact with the graphics subsystem using the following graphics stream control messages and events.

- Context_Request – Tools use this message to request operations on shared Context for the collaborative session. The target context is specified as an argument, as are the control parameters. Graphics context control includes operations like creating, setting up, tearing down, and deleting session contexts. It implements manipulation of shared 3D graphics windows.

  At the receiving end, the session subsystem sends this event to the Mapper. The handler function performs the appropriate Context control operations. A Context_Notify message is sent to the sender, with control information about the session context.

- Context_Notify – The session subsystem sends this event to the Mapper, typically when it receives this message in response to a Context_Request. It specifies the target context and control information. The handler function allows tools to take appropriate action.

- Stream_Request – Tools use this message to request operations on shared graphics interaction streams for the collaborative session. The target context and stream are specified as arguments, as are the control parameters. Graphics

stream control includes operations like transporting models, setting up viewing and modeling parameters, transmitting dynamic viewing information and control parameters for the very process of interacting in a 3D window etc.

At the receiving end, the session subsystem sends this event to the Mapper. The handler function performs the appropriate Stream control operations. A Stream_Notify message is sent to the sender, with control information about the graphics interaction stream.

- Stream_Notify – The session subsystem sends this event to the Mapper, typically when it receives this message in response to a Stream_Request. It specifies the target context and stream, as well as control information. The handler function allows tools to take appropriate action.

Sha-Poly and Sha-Draw are collaborative tools that have been implemented on this substrate. They are described in Appendix A.

The collaborative graphics substrate provides its functionality *via* the abstraction of 2D and 3D Graphics Widgets. These widgets encapsulate the underlying messaging, and transparently provide collaborative functionality for shared graphical interaction. 2D and 3D graphics features are incorporated in tools by creating and manipulating these widgets. They support the notion of callback functions to allow tools to take actions when different events occur.

3.2.5 Portable Multimedia

This abstract multimedia system provides access to available hardware audio and video facilities on a workstation in a device-independent manner, providing source code level compatibility across multiple platforms. It encapsulates details of media format and device specific interaction, providing a high level abstraction for development of multimedia tools. The current implementation supports Sun, SGI and HP workstations.

Tools interact with the media subsystem using the following media control messages and events.

- Control_Request – Tools use this message to request operations of the actual media system *via* the media context. The target context is specified as an argument, as are the control parameters. Multimedia control includes operations activating audio and video hardware, controlling input and output volume, or sampling frequency, and other general media control mechanisms.

  At the receiving end, the media subsystem sends this event to the Mapper. The handler function performs the appropriate media control operations, *via* the media substrate. A Control_Notify message is sent to the sender, with control information about the media context.

- Control_Notify – The media subsystem sends this event to the Mapper, typically when it receives this message in response to a Control_Request. It specifies the target context and control information. The handler function allows tools to take appropriate action, like updating the user interface etc.

3.2.5.1 Audio

The audio component of the device independent multimedia library provides a high level abstraction of the underlying physical device. It consists of a suite of device-specific libraries. Each audio system supported is represented by a library in the suite. All libraries implement the same audio paradigm and present the same application programming interface. This API presents the notion of an abstract audio device, permitting maintenance of source-level portability across several systems in application programs. It allows programmatic control of capture and playback of audio information from the hardware. It provides mechanisms for controlling sampling frequency, sample size, external device output volume and input sensitivity. Audio data is maintained in a portable, device-independent format for transport. It is translated to a device specific form before actual playback.

3.2.5.2 Video

The video component of the device independent multimedia library provides a high level abstraction of the underlying physical device. It consists of a suite of device-specific libraries. Each video system supported is represented by a library in the suite. All libraries implement the same video paradigm and present the same application programming interface. This API presents the notion of an abstract video device, permitting maintenance of source-level portability across several systems in application programs. It allows programmatic control of capture and playback of video information from the hardware. It provides mechanisms for controlling frame frequency, frame size, and color issues. Video data is maintained in a portable, device-independent format for transport. It is translated to a device specific form before actual playback.

3.2.6 Collaborative Multimedia Substrate

It is based on the structural model and uses the development substrate and the multimedia substrate to implement device independent distributed and collaborative multimedia. It enables incorporating multimedia features and facilities into tools, and supports collaborative multimedia interaction.

Tools interact with the media subsystem using the following media stream control messages and events.

- Context_Request – Tools use this message to request operations on shared Context for the collaborative session. The target context is specified as an argument, as are the control parameters. Multimedia context control includes operations like creating, setting up, tearing down, and deleting shared contexts for multimedia interaction. It implements manipulation of video windows and audio contexts.

  At the receiving end, the session subsystem sends this event to the Mapper. The handler function performs the appropriate Context control operations. A

Context_Notify message is sent to the sender, with control information about the session context.

- Context_Notify – The session subsystem sends this event to the Mapper, typically when it receives this message in response to a Context_Request. It specifies the target context and control information. The handler function allows tools to take appropriate action.

- Stream_Request – Tools use this message to request operations on shared multimedia streams for the collaborative session. The target context and stream are specified as arguments, as are the control parameters. Multimedia stream control includes operations like transporting data, setting up viewing parameters, transmitting transport control information, and control parameters for the very process of interacting with the media.

  At the receiving end, the session subsystem sends this event to the Mapper. The handler function performs the appropriate Stream control operations. A Stream_Notify message is sent to the sender, with control information about the multimedia interaction stream.

- Stream_Notify – The session subsystem sends this event to the Mapper, typically when it receives this message in response to a Stream_Request. It specifies the target context and stream, as well as control information. The handler function allows tools to take appropriate action.

Sha-Phone, Sha-Video, and Sha-Talk are conferencing tools that have been implemented on this substrate. They are described in [8, 10].

The collaborative multimedia substrate provides its functionality *via* the abstraction of Audio and Video Widgets. These widgets encapsulate the underlying messaging, and transparently provide collaborative functionality for shared multimedia interaction. Audio and video features are incorporated in tools by creating and manipulating these widgets. They support the notion of callback functions to allow tools to take actions when different events occur.

3.3 Tools

Shastra Tools are the building blocks of the runtime system. Kernels and Session Managers are management tools, responsible for maintaining the distributed and collaborative environment. Brokers offer distribution and task brokering facilities. Toolkits implement scientific design and manipulation functionality, and Service Tools provide mechanisms for communication and animation.

3.3.1 Kernels

The Shastra Kernel is responsible for maintenance of the runtime environment. It consists of a group of cooperating Kernel processes. It maintains information about all instances of tools in the distributed system, and keeps track of all environment relevant activity. A Directory facility lets users dynamically discover what tools (and users) are active in the environment at any time, as well as what functionality they provide. This includes information about ongoing collaborative sessions and their membership. A Location facility provides contact information about where the tools are running, letting tools dynamically connect to each other to access functionality. A Routing facility enables transport of data and control information between tool instances. The Kernel supports the following environment maintenance requests.

- Register – Request to become part of the runtime environment.
- Unregister – Request to leave the Shastra environment.
- Terminate – Request to cause the termination of another tool.
- Message – Request to send a data or control message to another tool.
- Session Start – Request to instantiate a new collaborative session.
- Join – Request to join an ongoing collaborative session.

A Kernel process executes at a well known port on all active hosts. Kernel processes communicate with each other and exchange Directory, Location and Routing

information. All Shastra tools on a host Register with the Kernel at startup, and become part of the runtime environment. They can access environment information *via* the Kernel, and use its facilities. They provide information about availability of audio, video and graphics hardware on their hosts and their ability to interact with it. The Unregister message is used when tools are ready to leave the environment, typically before termination. Abnormally terminated tools are unregistered automatically. Tools can cause the termination of other tools by using the Terminate message, if they have the appropriate capability. Tools that are not directly connected can send multimedia messages to one another using the Message request. Messages are routed *via* the Kernels. Collaborative sessions are started using the Session Start message. Tools can request assimilation into an ongoing collaborative session by sending the Join message to the Kernel. The Kernel routes the message to the appropriate Session Manager. Tools can also invite other tools tools to participate in the collaborative activity by sending the Invite message to the Kernel. The Kernel routes the message to the appropriate tool.

3.3.2 Brokers

Brokers are specialized Service Tools in the Shastra environment. Brokers create many instances of server tools for the different services offered in the environment. This occurs automatically or under control of tools using the Broker. In the simple case a Broker behaves as a surrogate client. Tools send multiple service requests to a Broker that uses its set of connected servers to service the multiple requests, and sends the results back to the client tools. All Brokers service the following brokering control requests.

- Start – Request to start a server tool.
- Stop – Request to terminate a server tool.
- Connect – Request to connect across the network to an existing server tool.
- Disconnect – Request to terminate a server connection.

- Service – Request to service a computational task.

Tools use the Start message to control creation of server instances, and the Stop message to terminate them. They can choose the server instances to be used for the task by sending Connect and Disconnect requests to the Broker. They use the Service message to request the Broker to perform operations on their behalf. The Broker forwards the request to an appropriate server.

In a more complicated scenario, tools send large computational tasks to application-specific Brokers which partition them, in a task-dependent manner, into independent subtasks. These subtasks are then serviced using the connected pool of server tools. The results are put together and transmitted to the requesting tools. Application-specific Brokers are created from the basic Broker, by extending the message set to understand new requests.

Brokers exploit tool-level cooperation to perform multiple tasks in parallel in a distributed setting, by harnessing the computational power of clusters of idle workstations on a network. They use load balancing criteria to optimize computation time of large tasks that can be decomposed into independent subtasks.

3.3.3 Session Managers

Collaborative Sessions, or Sessions, are instances of synchronous multi-user collaborations or conferences in the Shastra environment. A collaboration in Shastra consists of a group of cooperating tools regulated by a Session Manager, the conference management tool of Shastra. One Session Manager runs per collaborative session. It maintains the session and handles details of connection and session management, interaction control and access regulation. It keeps track of membership of the collaborative group, and serves as a repository of the shared objects in the collaboration. It supports a multicast facility needed for information exchange in a synchronous multi-user conferencing scenario. It has a constraint management subsystem that resolves conflicts that arise as a result of multi-user interaction, enabling maintenance of mutual consistency of operations. It has a regulatory subsystem that

controls synchronous multi-party interaction, and provides a floor control facility based on turn–taking. Every Session Manager implements functionality to service the following session control requests.

- Invite – Request to invite a tool to an ongoing session.
- Join – Request to join an ongoing session.
- Remove – Request to remove a tool from a session.
- Leave – Request to leave a session that the tool is a member of.
- End – Request to terminate a collaborative session.

It also serves the following interaction control requests.

- Format – Request to set session format.
- Capabilities – Request to set access regulation capabilities.
- Interaction Mode – Request to set interaction mode for the session.
- Request Floor – Request to get floor control for the session.
- Release Floor – Request to release floor control for the session.
- Assign Floor – Request to assign floor control for the session.

A collaborative session in Shastra is started by a tool when it sends the Session Start message to the local Kernel. This causes the instantiation of a Session Manager for the incipient session. The initiating tool becomes the Session Leader. A tool sets session format using the Format message. Sessions may be Formal, where participation is by invitation only, or Informal where any tool can dynamically join the conference. The Leader assigns capabilities of other participants for collaborative activity in the session using the Capabilities message. The interaction mode for a session is specified using the Interaction Mode message. Interaction can occur in

the Regulated or Free mode. In the Regulated mode, tools request and relinquish the floor using Request Floor and Release Floor messages. The leader can explicitly assign floor control using the Assign Floor message. In the Free mode, interaction is regulated *via* capabilities assigned to session participants. Capabilities are described in a later section. Other tools are invited to participate in a session by sending them the Invite request *via* the Kernel. Tools can dynamically join ongoing sessions by sending the Join message to the relevant Session Manager *via* the Kernel. The Session Manager uses session format information to control dynamic incorporation of tools. The Leader can remove a participating tool from the session using the Remove message. Tools can discontinue participation in the session by sending the Leave message to the Session Manager. A session is terminated by the Leader using the End message.

Application-specific Session Managers for different collaborative tasks are created from the basic Session Manager that provides application independent connection, communication and collaboration control facilities. Such session managers support additional messages for collaborative operations specific to the application.

3.3.4 Fronts

Front End or Front is the term used to collectively refer to all tools in the Shastra environment that a user directly interacts with. This includes Toolkits – actual engineering and design tools, Services – special purpose tools primarily for communication, and Games – recreational tools. Fronts are created by specializing the basic Front End which is a minimal collaboration-aware tool which understands Shastra protocol messages, and generates requests to interface with the Kernel, Session Managers, Brokers and standard Services. Figure 3.3 shows a view of the Shastra world, where different tools interact to support a collaborative environment.

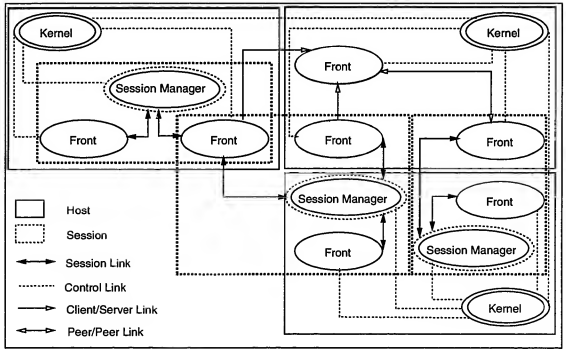

Figure 3.3 Information Flow in the Shastra Environment

Fronts add core functionality to the Application Engine. They extend Network Interface signatures to understand new requests, and to offer different services in the environment. This makes it possible for Fronts to connect to each other in client/server and peer/peer settings to access functionality, and to exchange data.

3.3.5 Toolkits

Currently Ganith, Shilp, Vaidak, Bhautik, Splinex and Rasayan are scientific Toolkits under the Shastra umbrella. They inter-operate to permit concurrent engineering and distributed problem solving. They are described in detail in [13], and are presented in a later section.

3.3.6 Services

The current set of Services contains communication and animation tools. In keeping with the Shastra philosophy of application-level cooperation, Services provide access to their functionality to other tools. Service tools behave as interface agents that understand different media types. Application developers are shielded from low-level manipulation of devices and media formats, This builds an abstract media-rich substrate for the design of sophisticated collaborative applications, since it enables use and transport of multimedia information. In the scientific setting, especially in design and analysis, a lot of the information shared by a collaborating team is expressed using structured 3D graphics. Inclusion of facilities for text, image, audio and video communication has greatly enhanced the quality of interaction, enabling more effective cooperation.

Sha-Talk, Sha-Phone, Sha-Video, Sha-Draw and Sha-Poly are services that are currently provided by the Shastra environment. They are described in detail in [8, 10].

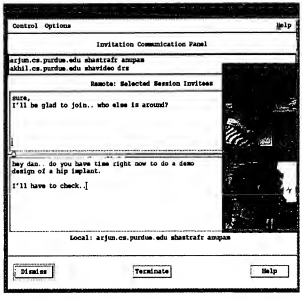

Figure 3.4 Multimedia Communication Support for Session Initiation

3.4 Runtime Environment

Figure 3.3 depicts the composition of the runtime system. The environment typically consists of a collection of instances of Kernels, Session Managers, and Brokers, as well as Fronts (Toolkits and Service Applications). The tools interoperate to support a dynamic collaborative environment.

3.4.1 Communication & Session Initiation

Fronts use the Directory and Location facilities of the Kernel, in conjunction with messaging facilities of the system to set up multimedia communication panels. By default, the panel allows textual conversations. The environment keeps track of availability of audio, video and graphics facilities on hosts. This lets tool users configure the panel appropriately, if higher level facilities can be used. The control panels drive Service Tools to conduct conversations.

The communication facilities of the environment are of great utility in the session initiation procedure. A Session is started by a user of a tool by sending a Session Start request. The Kernel instantiates a Session Manager. The session creator becomes the session leader, and is automatically inducted into the conference. He queries the system to discover other active users in the environment, and invites them to join the session using the Invite request. The invitees are prompted *via* their Kernels for participation in the session, and a communication panel is created at each invited site. The leader can broadcast to the invited group, or communicate with individual invitees, through his communication panel. The invitation specifies the capabilities the invitee will have when he joins. Communication permits rapid convergence to acceptance or declining of the invitation. Invitees that accept are inducted into the conference. (Invitees can be participants in multiple conferences simultaneously). Figure 3.4 depicts a session invitation control panel. Users of tools query the environment to discover sessions of interest, and use the Join message to request admittance. The request is routed to the session leader *via* Kernels and the Session Manager. If session format is Informal, where any tool is allowed to join the "open" session, the remote tool is incorporated into the conference by the Session Manager. For Formal "closed" sessions, a communication panel is created to enable negotiation between the requester and the leader, who may allow the remote tool to be assimilated into the conference.

3.4.2 Collaborative Interaction

Collaborative interaction in Shastra occurs in two modes. In the Regulated mode, which is based on baton passing, tools request control of the "floor", the collaboration context, using the Request Floor message. This results in a Master-Slave interaction, where all interaction occurs at the Master site that has the baton, and appropriate information is relayed to all Slave sites. Application developers utilize the broadcast facility of the Session Manager to distribute the input of low computation tasks and the output of high computation tasks to benefit from the distributed setting.

Baton passing itself may be Non-Preemptive, based on voluntary relinquishment of the baton, or Preemptive. Users relinquish the floor using the Release Floor message. The Regulated mode is useful to simulate blackboarding, demonstration and walk-through scenarios, as such interactions are naturally expressed using a turn taking mechanism. The leader can explicitly assign turns using the Assign Floor message.

In the Free interaction mode, activity is regulated *via* capabilities assigned to session participants by the leader. The Capabilities message is used to dynamically control capabilities of session participants in this truly collaborative setting, which allows simultaneous multi-participant interaction. The collaboration infrastructure of Shastra has a two-tiered regulatory subsystem used to control dynamic interaction. Site-based capabilities control the interaction of a user in the context of the collaboration. Shastra collaborations support the following capabilities.

- Access – controls whether or not a user sees shared context and collaborative interaction.
- Browse – regulates whether or not a user has independent local viewing control.
- Modify – regulates whether or not a user can modify shared state.
- Copy – controls whether or not a user can copy shared objects.
- Grant – controls whether or not a user can perform session regulation operations.

The Access capability is roughly equivalent to a "Read" permission for the shared collaboration context, *e.g.* a shared window. It is useful in scenarios requiring some form of hiding. The Browse capability governs browsing and independent local viewing of the shared context. It is roughly equivalent to an "Execute" permission. The Modify capability, which is roughly equivalent to a "Write" permission, controls whether or not a participant can modify the state of the collaboration by introducing objects to the session, or by interacting with shared objects. The Copy capability regulates copy propagation of shared objects in the collaboration. The Grant capability defines whether a user can set site capabilities or invite users to join the session.

Object permissions are application specific, though they are generally modeled after the site capabilities. They regulate what actions the participant can take on shared objects. Users specify Access, Browse, Modify and Copy permissions for shared objects, to control whether other users can Read, Write, Execute and Copy shared objects that they introduce to the collaborative session. In general, the more restrictive of Site Capability and Object Permission applies for shared objects.

Access regulation information is maintained in the Session Manager. Site capabilities are regulated by the leader or a user with the Grant capability. Object permissions are regulated by the owners of collaboration objects. Different capability and permission settings for participants and objects generate a variety of interaction modes at runtime. *E.g.* giving all participants only Access capability and one participant Modify capability, effectively simulates a turn-taking situation, where one participant alters the state of the collaboration, and everyone else observes the results. Adding Browse capability at all sites results in a flexible Master-Slave situation, with every site capable of independent local views. Allowing everybody to Modify the state of the collaboration creates a free interaction situation. Similarly, setting permissions for an object regulates the operations a participant can perform on it.

3.5 Computation Model

Shastra collaborations are implemented using a Hybrid Centralized-Replicated computation model. A central Session Manager regulates collaborative activity of multiple tool instances. The core part of a Session Manager communicates with an abstract Collaboration Slave in the tool via Shastra Network Interfaces. The Slave maintains shared state and context and performs collaboration-relevant operations under directives from the Session Manager. The application-specific parts of the Session Manager and the tools inter-operate in a similar manner. This method of operation is termed as Proxy shared window management. The architecture of a typical collaborative session in Shastra is depicted in Figure 3.5.

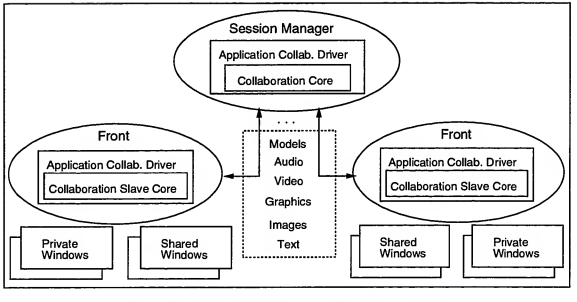

Figure 3.5 Architecture of a Collaborative Session

3.5.1 Replication

In the hybrid computation model for the multiple user system, a copy of the tool runs at each site involved in the collaboration. In the scientific setting, where 3D graphics is used to express a lot of information, the replicated Front instances provide performance benefits, since no central site is weighed down by view-generation computation. The replication scheme supports heterogeneity – the Session Manager communicates with Fronts at an abstract level, and doesn't concern itself with details of how the Front actually executes its directives. In a heterogeneous environment, this is a big win, because replication allows Fronts to execute on a variety of hardware platforms. Another benefit of the replication scheme with Proxy shared window management is that it intrinsically supports the notions of Private and Shared workspaces and interaction. Only activity in the collaboration context (windows and work areas) is regulated by the Collaboration Slave part of a tool, and is shared between participants. Local windows isolate private interaction. Fronts provide regulated mechanisms for moving work between private and shared workspaces. The underlying notion is to provide a non-intrusive environment for collaborative work, where users can choose to perform some tasks in private contexts, and some in shared ones.

In the simple case, Session Managers deal with multiple, functionally identical Fronts. We call this kind of situation a Homogeneous Collaboration. In a more complicated scenario, functionally different Fronts collaborate over mutually relevant tasks. Though this increases the complexity of the Session Manager, which cannot rely any more on the assumption of identical contexts, Heterogeneous Collaborations are no harder to implement, since Session Managers and Fronts inter-operate at an abstract level.

3.5.2 Centralization

Collaborations between Fronts in Shastra are regulated by a central Session Manager that is responsible for maintaining and regulating collaborative activity. It serves

as a repository for shared objects in the collaboration. The centralization of collaboration state in the Session Manager makes it convenient and efficient to bring late joiners up to date when they are assimilated into the session. The Session Manager serves as a context for regulation of interaction during collaborations. Importantly, serialization of input from all participating sites is conveniently performed in the Session Manager.

The Session Manager also performs constraint management to maintain consistency and regulate mutually conflicting tasks. In collaborative interaction different participants can attempt contradictory modifications to the state of the collaboration. Session Managers can identify and deal with these inconsistencies, which are inherent to synchronous and distributed multi-party interaction. The current method involves registering constraint databases for operations, and dynamically checking operations for constraint violations. Fronts that have actions denied due to constraint violations are notified appropriately, using available messaging facilities. In Heterogeneous Collaborations, between instances of different toolkits, constraint management involves dealing with the coupling of related data structures and operations for mutual consistency.

3.6 CSCW Environments

The Shastra system is targeted towards two main problem solving scenarios. The first involves synchronous conferencing of collaboration aware tools. We call this a Collaborative problem solving scenario. The other case involves problem solving using a mix of collaboration aware services and collaboration-unaware but cooperative tools. We call this a Quasi-Collaborative problem solving scenario.

3.6.1 Distributed Multimedia

The distributed multimedia system is at the core of the collaborative environment. It is built around multimedia Agents that provide device independent handling of multiple media types. Agents are built on top of the Shastra layer which is a connection,

communication, and collaboration substrate. The connection layer provides connection setup and maintenance facilities over a network, and implements the protocols needed in the system. The communication layer supports platform independent data exchange between connected agents. The collaboration layer provides session management and regulation facilities. In this system, agents are cooperation aware both at the application level and at the user level. They provide 1-1 personal communication services for spontaneous cotemporal interaction, as well as asynchronous multimedia messaging.

These agents are used to build multimedia conferencing facilities in a distributed setting by specifying behavior in the connection, communication and collaboration layers. They use application level communication protocols for 1-1, 1-n and n-n interaction using the different media types.

3.6.2 Collaborative Problem Solving

This involves the creation of synchronously conferenced systems built from scientific toolkits. The process involves adoption of Shastra architectural guidelines in the toolkits by building them in accordance with the requirements, or by wrapping them in functionality that provides the same abstract architecture. Session Managers are then created from the core Session Manager to deal with the interaction and collaboration details of the specific task. This includes the specification of a constraint management subsystem to deal with inconsistencies arising from multi-user interaction. Application conferencing provides a facility for shared manipulation of application-specific objects. Shastra Services are used for colocation to aid the collaborative effort. This provides support for Text Conferences, Audio and Video conferences, Collaborative 2D Sketching, and 3D Visualization.

3.6.3 Quasi-Collaborative Problem Solving

This scenario is more suited to tools that will be hard to conference synchronously due to their complexity or due to intractability in the architectural paradigm of

Shastra. In these problem solving situations, Shastra facilities are used for colocation to aid in the problem solving effort as well as in the review and analysis phase. The Shastra architecture facilitates Client-Server and Peer-Peer mode interconnections between systems for collaborative problem solving.

4. THE SYSTEM AND APPLICATIONS

4.1 Runtime System

4.1.1 Introduction

Multimedia workstations have become commonplace because of recent advances in electronics, computer and communication technology. Current audio, video, and graphics processor architectures, coupled with high speed networking and compression techniques, have presented us with a very powerful tool – the desktop system. Computer mediated mechanisms built on top of these systems provide us with the means to exchange multimedia information, and will revolutionize how we collaborate in the scientific setting. This is especially true in the scientific domain, where problem solving is cooperation-intensive.

Much recent research and developmental effort has been directed towards multi-user systems. Our goal is to depart from traditional single-user scientific manipulation systems to use computing and multimedia technology to build multi-user (collaborative) scientific design and analysis environments. The objective is to develop the next generation of scientific software environments where multiple users, *e.g.* geographically distributed collaborative engineering design teams, create, share, manipulate, analyze, simulate, and visualize complex three dimensional geometric designs over a heterogeneous network of workstations and supercomputers. Scientific CSCW would benefit greatly from applications with shared drawing and viewing surfaces that support content dependent sharing – the applications are collaboration-aware, and support simultaneous multi-user manipulation of application-specific objects. This

would add a new dimension to the kind of cooperation that can occur in collaborative problem solving, because it would permit cooperative manipulation and browsing of objects in the context of applications that manipulate them.

We have adopted the approach of integrating a collection of function-specific tools into a distributed and extensible environment, where tools can easily use facilities provided by other tools. Isolation of functionality makes the environment modular, and makes tools easy to develop and maintain. Distribution lets us benefit from the cumulative computation power of workstation clusters. Tool-level cooperation allows us to exploit the commonality that is inherent to many scientific manipulation systems. An enabling infrastructure of communication and interaction tools, display and visualization facilities, symbolic processing substrates, and simulation and animation tools saves avoidable re-implementation of existing functionality, and speeds up the application development process.

The collaborative scientific environment provides mechanisms to support a variety of multi-user interactions spanning the range from demonstrations and walk-throughs, to synchronous multi-user collaboration. In addition, it facilitates synchronous and asynchronous exchange of multimedia information that is useful to successfully communicate at the time of design, and to share the results of scientific tasks, and often necessary to actually solve problems. The infrastructure provides facilities to distribute the input of low computation tasks – to obtain the parallelism benefit of distribution, and the output of compute intensive tasks – to emphasize sharing of resources among applications. It provides a convenient abstraction to the application developer, shielding him from lower level details, while providing him with a rich substrate of high level mechanisms to tackle progressively larger problems.

4.1.2 Scientific Manipulation Environments

Shastra is an extensible, distributed and collaborative geometric design and scientific manipulation environment. At its core is a powerful collaboration substrate – to support synchronous multi-user applications, and a distribution substrate – that

emphasizes distributed problem solving for concurrent engineering. Shastra provides a framework for distribution, collaboration, session management, data sharing and multimedia communication along with a powerful numeric, symbolic and graphics substrate. It enables rapid prototyping and development of software tools for the creation, manipulation and visualization of multi-dimensional geometric data.

The Shastra environment consists of multiple interacting tools. Some tools implement scientific design and manipulation functionality (the Shastra Toolkits). Other tools are responsible for managing the collaborative environment (Kernels and Session Managers). Yet others offer specific services for communication and animation (Service Applications). Tools register with the environment at startup, providing information about the kind of services that they offer (Directory), and how and where they can be contacted for those services (Location). The environment supports mechanisms to create remote instances of applications and to connect to them in client-server or peer-peer mode (Distribution). In addition, it provides facilities for different types of multi-user interaction ranging from master-slave blackboarding (Turn Taking) to synchronous multiple-user interaction (Collaboration). It implements functionality for starting and terminating collaborative sessions, and for joining or leaving them. It also supports dynamic messaging between different tools. Tools are thus built on top of the abstract Shastra layer, which is depicted in Figure 4.1. The Shastra Layer is a connection, communication and collaboration management substrate. Shastra tools inter-operate using facilities provided by this layer.

4.1.3 System Features

The Shastra architecture is described in detail in [8]. The scientific toolkits are presented in [13]. Here, we present salient features. The design of Shastra is the embodiment of a simple idea – scientific manipulation toolkits can abstractly be thought of as objects that provide specific functionality. These objects exchange messages, automatically or under user command, to request other objects to perform operations.

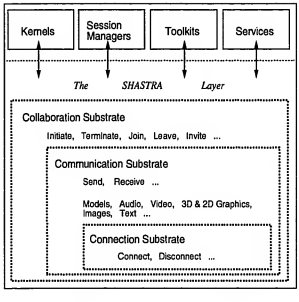

Figure 4.1 The Shastra Layer

At the system level, Shastra specifies architectural guidelines and provides communication facilities that let toolkits cooperate and exchange information to utilize the functionality they offer. At the application level, it provides collaboration and multimedia facilities allowing the development of applications in which users cooperate to solve problems. A synergistic union of these two ideas lets us design sophisticated problem solving virtual machines.

The high level block architecture of all tools in Shastra is depicted in Figure 3.1. This architecture makes it easy for tools to connect to other tools and request operations, synchronously as well as asynchronously. A tool has an application specific core – the Application Engine, which implements the core functionality offered by the tool. Above the core is a Functional Interface Mapper that invokes functionality embedded in the Engine in response to requests from the the Graphical User Interface, ASCII Interface or the Network Interface. The GUI is application specific. The ASCII interface is a shell-like front end for the application. Tools communicate with other tools in the environment, *via* the Shastra substrate, through an abstract Network Interface that multiplexes multiple simultaneous network connections and implements the Shastra communication protocol [8].

The entire set of connected Network Interfaces of Shastra tools implements the abstract Shastra layer at runtime (see Figure 4.1). It maintains the collaborative environment, provides access to functionality of different systems, and provides facilities for initiating, terminating, joining, leaving and conducting collaborations.

4.1.4 Tools

Shastra Tools are the building blocks of the runtime system. Kernels and Session Managers are management tools, responsible for maintaining the distributed and collaborative environment. Shastra Toolkits provide scientific design and manipulation functionality, and Service Tools provide mechanisms for communication and animation. Toolkits and Service Tools are collectively referred to as Front Ends, or simply Fronts, since they are the actual sites of user interaction. Any Front can access the

Shastra environment to instantiate tools locally or on remote sites, and to terminate previously instantiated tools. Fronts can connect directly to each other to exchange data in client-server or peer-peer settings using the Shastra substrate.

4.1.4.1 The Kernel

The Shastra Kernel is responsible for maintenance of the runtime environment. It keeps track of all instances of tools in the distributed system. It consists of a group of cooperating Kernel processes. A Directory facility lets users dynamically discover what tools are active in the environment at any time. A Location facility provides contact information about where the tools are running, letting applications dynamically connect to each other to access functionality.

4.1.4.2 Session Managers

A Session Manager is a management tool in the Shastra environment. It maintains a collaborative session and handles details of connection and session management, interaction control and access regulation. It is a repository of the shared objects in a collaboration, and keeps track of membership of the collaborative group. A collaborative session in Shastra is started by a user through a Front. One instance of a Session Manager runs per collaborative session. The Session Manager provides a multicast facility needed for information exchange in synchronous multi-user conferencing. It has a constraint management subsystem that resolves conflicts that arise as a result of multi-user interaction, enabling maintenance of mutual consistency of operations. It also provides a floor control facility based on baton–passing. The architecture of a typical collaborative session in Shastra is depicted in Figure 3.5. Figure 3.3 shows a view of the Shastra world, where different tools interact to support a collaborative environment. The Shastra collaboration architecture uses a replicated computation model for the multiple user system – a copy of the application (the Front) runs at each site involved in the collaboration. The main benefits derived from this replication are heterogeneity and performance.

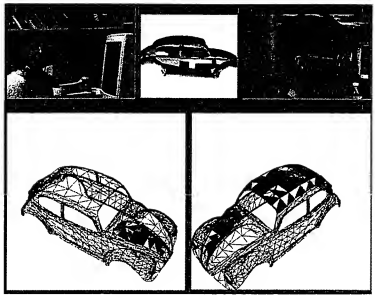

Figure 4.2 Collaborative Polyhedron Smoothing using Shilp and Ganith

4.1.4.3 Toolkits

Currently Ganith, Shilp, Vaidak, Bhautik, Splinex and Rasayan are scientific tools under the Shastra umbrella. These toolkits are powerful standalone systems that operate on application-specific models. They have been integrated into the Shastra environment, and permit concurrent engineering and distributed problem solving by providing access to their functionality to other toolkits. This interoperability enhances the functionality of each toolkit.

The Ganith algebraic geometry toolkit manipulates arbitrary degree polynomials and power series [8]. It is used to solve a system of algebraic equations and visualize its multiple solutions. It incorporates techniques for multivariate interpolation and least-squares approximation to an arbitrary collection of points and curves, and C^1-smoothing using low-degree implicit patches. Other Shastra toolkits use the algebraic manipulation capability it provides at its network interface – curve and surface intersection, interpolation, and approximation functionality.

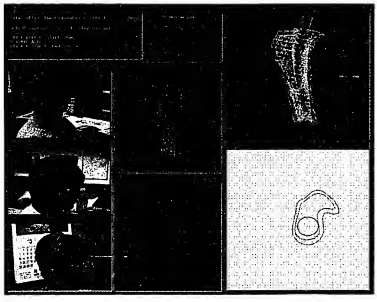

Figure 4.3 Collaborative Custom Hip Implant Design – Contour Generation

Splinex is a curve and surface modeling toolkit that provides interactive creation and manipulation of implicit and parametric splines in Bernstein-Bezier and A-spline bases [8]. It provides Bezier and A-spline surface manipulation capability in the environment.

Shilp is a boundary representation based geometric modeling system [8]. Current functionality of the toolkit includes extrude, revolve and offset operations, edit operations on solids, pattern matching and replacement, boolean set operations, fleshing of wireframes with smooth algebraic surface patches, and blending and rounding of solid corners and edges. These operations can be invoked by local users and by remote toolkits. Figure 4.2 shows a site during collaborative polyhedron smoothing in Shastra using Shilp and Ganith. Conferenced Shilp instances use multiple remote instances of Ganith to interpolate faces of a polyhedral car model in parallel, to produce a curved surface model with C^1-continuous surface patches. The toolkits communicate *via* their network interfaces, and Ganith services Shilp requests. The original

polyhedral car model (top-center), one designer's part of the shared task (bottom-left) and the shared, partially complete curved surface car model (bottom-right) are shown. Images of a supporting video-conference are also shown.

The Vaidak Medical Image Reconstruction Toolkit is used to construct accurate cross-sectional, surface and solid models of skeletal and soft tissue structures from CT (Computed Tomography), MRI (Magnetic Resonance Imaging) or LSI (Laser Surface Imaging) data. These models can be used by Shilp for design activity, and by Bhautik for physical simulations. In a distributed problem solving scenario, a geometric model of a human femur is reconstructed in Vaidak and manipulated in Shilp. In Figure 4.3, a designer uses Shilp to interactively create a geometric model of a hip implant (right-top), by generating cross-sectional contours of the implant (bottom-center and right) from a sectional model of the femur (center) created in Vaidak. A video conference is used for communication.

The Bhautik physical analysis toolkit provides mesh generation facilities and a graphics interface to set up, perform and visualize physical simulations on geometric models created interactively using Shilp, or models reconstructed from imaging data by Vaidak. Figure 4.4 shows a load transfer finite element analysis used in custom design of hip implants [8], using Vaidak, Shilp and Bhautik toolkits.

Rasayan can compute and visualize the "docking" of drug and protein molecules under molecular Brownian motion. It provides mechanisms for analysis and visualization of the potential energy surfaces of the molecules and the stationary points on these surfaces.

4.1.4.4 Services

The current set of Shastra services contains communication and animation tools. The objective is to provide a media-rich communication substrate for the design of multimedia applications, by relieving application developers of the burden of low-level manipulation of devices and media formats. In the scientific setting, especially in design and analysis, most of the information shared by a collaborating team is oriented

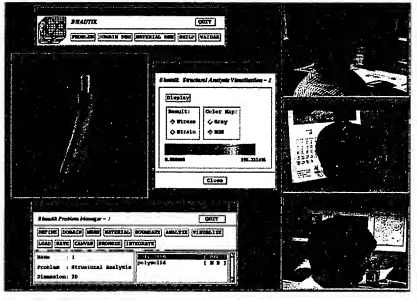

Figure 4.4 Stress Analysis Visualization in Collaborative Custom Implant Design

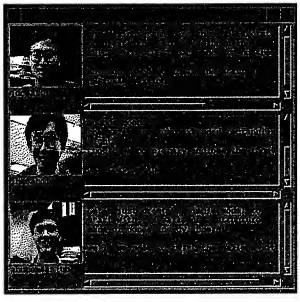

Figure 4.5 One Site in a Three-Way Text Conference using Sha-Talk

towards structured 3D graphics, and is typically application specific. However, inclusion of facilities for text, image, audio and video communication has greatly enhanced the quality of interaction, enabling the design of more sophisticated applications.

Sha-Talk is a text communication tool that supports synchronous n-way textual conversations. It is useful for designers who do not have multimedia communication facilities on the desktop. In Figure 4.5 we depict how Sha-Talk provides a simple textual conferencing facility that is especially useful when other communication methods are unavailable. Image bitmaps identify the owner of a text panel.

Sha-Draw is a Shastra environment sketching tool that facilitates the generation and display of simple 2D pictures using a rich set of primitive operations. A collaborative session consisting of Sha-Draw applications lets a group of collaborators synchronously create and edit simple 2D sketches on shared whiteboards. Figure 4.8 depicts one site in a collaborative sketching session.

Sha-Poly is a collaborative visualization and graphical-object browsing and manipulation tool. It supports shared viewing of 3D models using different display and

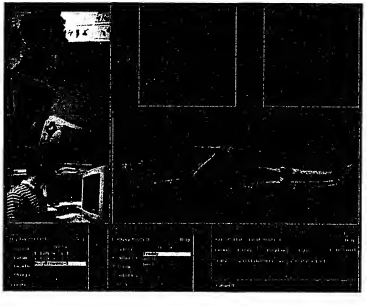

Figure 4.6 Shared Visualization of Volume Data using Vaidak and Sha-Poly

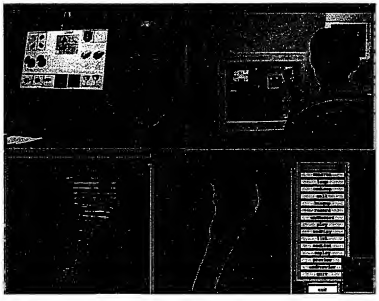

Figure 4.7 Video Support for Design – Visual Topological Verification

visualization techniques in a synchronously conferenced setting. Figure 4.6 shows one of three sites with independent private windows, and a shared conference window, for volume visualization of large medical data sets. Here a group of researchers uses Sha-Poly to share volume visualization images of a head with cutaways (top-center and right), and a cadaver (center). The images are generated by Vaidak from volume data.

The Sha-Phone service is used to record and playback audio information stored in multimedia objects. An n-way audio conference is conducted by setting up a collaborative session consisting of Sha-Phone instances.

Sha-Video handles image data (without sound) – both still images and motion video. It is used, both directly and by other tools, to playback and record video information stored in multimedia objects. A collaborative session consisting of Sha-Video applications provides the mechanism to conduct a silent video conference. In Figure 4.7 a researcher (top-right) uses a live video window (top-left) to confirm the topological accuracy of a reconstructed femur (bottom-left, bottom-right) in Vaidak.

Gati is an animation server that provides distributed and collaborative real-time interactive animation in two and three dimensions. The system supports a high level animation language based on a commands/event paradigm.

4.2 Collaborative Problem Solving

We now describe an example of multi-user cooperative design in the context of Shastra. We have built an application for collaborative set operation based design using Shilp and Sculpt, two Shastra toolkits. It permits a group of designers to cooperatively create a 3D model by performing set operations on simpler models in Shilp using Sculpt as a back end to perform the actual operations. Shilp is a geometric modeling system. Sculpt is optimized to perform set operations – Union, Intersection, Difference and Complementation on polyhedral geometric models [95]. We use Shastra's application level cooperation substrate and user level collaboration substrate to link the two applications. The result is a powerful multi-user interactive design facility.

4.2.1 Motivation

The design paradigm in Shilp emphasizes creation of complex models by performing operations on simpler models in a hierarchical fashion. Set operation based design (Constructive Solid Geometry) is a very flexible way of creating intricate 3D designs. Conventional systems support this method in the single user setting. A designer creates a design by going through the multiple steps involved. This application presents a departure from the traditional method – a collaborative design environment where a group of designers cooperatively create large designs. It provides facilities to enable cooperation between multiple users.

In boundary representation based solid modeling systems, generation of the results of set operations is compute intensive. Also, the design process can be represented as a tree in which the lower levels are often parallelizable – a group of designers can work independently on those parts. The application improves throughput of the

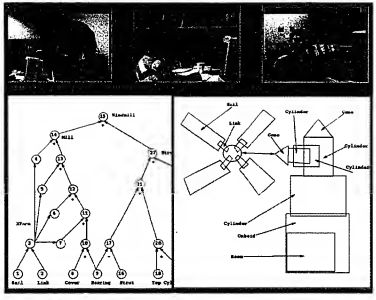

Figure 4.8 Using Sha-Draw for Shared 2D Sketching

design operation by providing a collaborative environment from the conception phase through the final design realization phase.

4.2.2 Startup Problem

One of the challenges of this design scenario is establishing a starting point. This issue is traditionally resolved by conducting a physical meeting where a design team is identified. Communication and information exchange between team members enables them to synchronize at a starting point, and gives them all an idea of the final design. Shastra eliminates the need for such a physical meeting by providing mechanisms to create a design group and media-rich support for a design brainstorming session. In Figure 4.8 Collaborating designers (top row) use shared windows to create a sketch (bottom-right) and a design graph (bottom left). In conjunction with a Sha-Video and Sha-Phone conference, Sha-Draw provides a powerful interaction environment.

A user running an instance of Shilp, queries Shastra to find out other active users in the environment. He uses the messaging facilities of Shastra to invite some of

them to a text conference, using Sha-Talk, and explores their interest in a particular design. Subsequently, he decides on his working group. Before the design process is started, some members of the team may have a mental picture of the object that they are attempting to design, while others may not. One of the designers initiates a collaborative brainstorming session using Sha-Draw, the multi-user sketching tool. If audio and video processing hardware is available, he invokes instances of Sha-Phone and Sha-Video, and initiates the relevant sessions. Sites without video hardware can use the software-only playback facility to display incoming video streams and generate outgoing streams. Audio-visual communication, provided by concurrent Sha-Phone and Sha-Video sessions (see Figure 4.8), leads to rapidly establishing the design goal. The group interactively creates a rough sketch of the intended design. Alternately, a stored image, or a live image of an actual physical object, or of its picture, can be broadcast to the group using Sha-Video. At the end of this phase, the entire team has a good idea of the task at hand.

The designers use Sha-Draw to set up the dependencies of various parts of the intended design in graphical form. A directed design graph, where nodes are solid models and edges are dependencies of the destination nodes, is subsequently created. The leaf (0 in-degree) nodes represent existing or primitive solid models, and internal nodes are intermediate models in the design process. A designated root node represents the final design goal. Directed edges indicate that the destination node is the result of an operation on all of the source nodes. Annotations in the graph indicate the operation needed to obtain the destination node from the source nodes (see Figure 4.8).

4.2.3 Design Outline

The operation is performed in two phases – design graph generation and model computation. The design graph, which is created in a Sha-Draw collaboration in the context of a sketch or video image of the final model, is a succinct summary of the entire design task. The image and/or the sketch is stored with the graph for

future reference. The graph is converted into a form amenable to this operation, with maximum in-degree of the nodes being 2, since only unary and binary operations are supported. An automatic DNF (Disjunctive Normal Form) decomposition is the simplest transformation, but doesn't produce the most efficient design graph. The design team cooperatively restructures the design graph, in a Sha-Draw collaboration, to meet the requirements.

In the model computation phase, a designer graphically positions models using the Shilp user interface. This is done to set up models in appropriate configurations for set operations. The actual operations to generate intermediate and final models of the design graph are performed in Shilp by automatically requesting geometric services from Sculpt.

4.2.4 The Shastra Setting

In a single user setting, a designer would compute the various nodes of the graph sequentially. The final model would be checked for goodness, and the computation process repeated till a satisfactory model was obtained.

In the multi-user setting, a collaborative Shilp session is initiated by one of the Shilp users in the environment. He specifies, to the local Kernel, the Shilp users who will be invited to participate in the session, and (by default) becomes the group leader. The Kernel instantiates a Session Manager, which starts a session with the group leader as its sole participant, and then invites the specified users of concurrently executing remote Shilp instances to participate. Users who accept are incorporated into the session. The Session Manager is responsible for providing access to shared objects and context at all participating sites.

Any participant can leave the session at any time, by simply de-linking from it. Users of other instances of Shilp in the environment can query the system to discover ongoing sessions, and request participation. The group leader regulates whether or not they are allowed to join the session. He can also invite other Shilp instances to participate in the session.

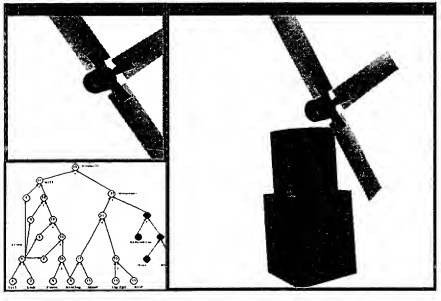

Figure 4.9 One Site in a Design Collaboration using Shilp and Sha-Draw

Every participating Shilp session creates two shared windows in which all the cooperative interaction occurs. More local windows can be created if desired. One shared window contains the design graph that is used to regulate the entire operation. The second window contains models as they are created or introduced to the session. Users introduce leaf node objects into the session by selecting them into the shared window.

The Session Manager partitions the design graph into slightly overlapping zones. The partitioning is based on the number of people in the collaborating group, and on the number of subtasks left in the operation (the number of uncomputed nodes in this case). It aims to minimize the number of shared nodes of partitions. Shared nodes in the graph are regions of contention in this collaboration scenario since they constitute dependencies in an otherwise parallelizable situation. The partitioning also aims to distribute the leaf nodes equally among the designers, since they usually represent nodes that have to be interactively created. It defines a scenario for fair, minimal-conflict cooperative interaction. The partitioning is dynamically altered as users join and leave the session. The group leader can explicitly specify and alter the partitioning. The partitioned design graph is displayed in a shared window, and serves as a context to regulate the collaborative operation, since it captures the state of the operation.

The partitioned zones are assigned to the collaborating designers, and are colored differently for identification. Every user is responsible for filling the intermediate nodes in his zone by first positioning the models on the incoming edges, and subsequently performing the actual set operation. This process is repeated till a satisfactory design is created. Figure 4.9 shows one site in the design of a simple windmill model. Here, the design graph (bottom-left) is used as a context to monitor progress and regulate the task. The designer sees the incomplete shared model (right) and the locally designed part (top-left). Figure 4.10 shows another site at the end of the operation. Here, the designer sees a completed shared model (left), the locally designed part (top-right) and the shared design graph (bottom-right).

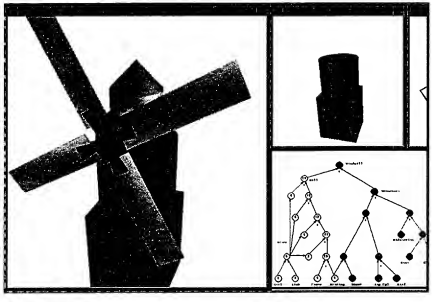

Figure 4.10 Another Site, at the End of a Collaborative Design Session

4.2.5 Collaborative Interaction

The Session Manager regulates all interaction relevant to the operation at participating sites. Interaction can occur in two modes.

In the Regulated mode, the user responsible for a zone creates all the models internal to the zone. All other users are denied access to the interior of a zone by the session manager. If the source nodes for an intermediate node are filled, a user locks the intermediate node by selecting it in the Graph Window. The session manager allows him access to models in the source nodes. The user interactively positions these models and performs the appropriate set operation, and the resulting model is assigned to the intermediate node, which is subsequently unlocked. At the boundary of a partition, users of adjacent zones must agree about the models at the boundary nodes so that inconsistencies are not created in the design.

A good group design protocol for this setting is to resolve boundary condition issues at the start of the operation, to prevent unnecessary cycles due to inconsistencies later on. This involves computing the subgraphs rooted at boundary nodes first, till satisfactory models at those nodes are obtained.

All operations are performed *via* the (central) session manager that is responsible for keeping all sites up-to-date, so that the users have a dynamically changing and continuously updated view of the operation in the shared windows – the design graph and intermediate models. Changing a node requires all of its dependencies to be re-evaluated. The operation is completed when all the nodes of the design graph have been evaluated. Any site with Copy permission can then extract the model from the session and save it.

4.2.6 Access Regulation and Collaboration Modes

The collaboration infrastructure of Shastra supports a two-tiered permissions-based access regulation mechanism. It is used to structure a variety of multi-user interaction modes at run-time. It allows a high degree of tailorability and flexibility in Shastra's CSCW applications in the domain of interaction as well as data sharing

and access control. The regulatory subsystem supports Access, Browse, Modify, Copy and Grant permissions for collaboration sites as well as for shared objects [8]. These permissions control what actions different users in the conference can take, and what objects they can operate on. In addition, tools can define and use new permissions for tool-specific actions. Permissions are controlled by the group leader or his designees. The regulatory subsystem also provides a mechanism to enforce and regulate floor control based on turn taking. Users can dynamically configure the interaction mode and permissions to suit the task.

In the brainstorming phase, for example, it is useful to allow everyone equal access to all operations and objects, to support free flow of ideas.

In the Unregulated mode for this operation, the partitioning merely suggests a minimal-conflict setting, and the session manager doesn't regulate interaction beyond what is specified by collaboration permission settings for the site and the object. In this mode a user can access any node if he has Access and Modify permissions for the collaboration.

The session manager allows only one user to manipulate a "hot spot" in the graph– where there is a possibility of contention – at any particular instant. It uses the first-come-first-served paradigm to decide which user gets temporary exclusive control. The last completed operation specifies the model associated with the node.

The baton passing facility of the system can be used for floor control – to take turns to set boundary nodes. Alternately, designers can use the auxiliary communication channels to regulate access, and to decide which users will set those nodes.

At one extreme, the Shastra implementation for collaborative design can be used by a single designer to design a solid model much like in a non-collaborative setting. Allowing other users to join the session with only Access and Browse permissions sets up the environment like an electronic blackboard to teach novice users the basics of the design mechanism. An appropriate setting of collaboration permissions and turn-taking can be used to allow hands on experience with the task. In conjunction

with the audio and video communication services of Shastra, this becomes a powerful instructional environment.

In a different situation, a group of n designers can set up an Regulated collaborative design session and collaborate to design an object. Each designer performs only the designated part of the shared design, and a speedup of as much as problem size / maximum partition size, can be achieved. Novice designers can join the ongoing session with only Access and Browse permissions, and get familiar with group dynamics of a collaborative session. In yet another situation, a group of n designers can start an Unregulated collaborative design session. Judicious use of the auxiliary communication facilities (Audio, Video and Text) to regulate design operations in a cooperative manner can let the team acquire a speedup factor of up to n.

4.2.7 Heterogeneity Issues

A Shastra conference consists of multiple tool instances at different sites. This localizes platform specific dependencies in the tool. It permits the Session Manager to view tools as high level application objects, without having to concern itself with low level details of how things are actually done. This approach supports the Shastra system on a wide variety of hardware platforms. Specifically, tools can take advantage of available hardware graphics facilities, video compression and decompression, and audio processing hardware. This greatly simplifies multimedia interaction management.

4.2.8 Collaborative Design

4.2.9 Collaborative Smoothing in Shastra

An example of multi-user cooperative design in the context of Shastra is Collaborative Smoothing using Shilp and Ganith toolkits. This permits a group of users to collectively smooth out a rough polyhedral model by fitting C1 or C0 continuous patches using Hermite interpolation [17]. Ganith is optimized to perform algebraic

manipulation – curve-curve, curve-surface, and surface-surface intersection, as well as interpolation [18]. Shilp is optimized for Boundary Representation based solid modeling. A coordinated nexus between the two applications lets us add a powerful design facility to the environment by drawing upon the functionality-sharing application level cooperation substrate, and the user level collaboration substrate of the Shastra environment.

4.2.9.1 Motivation

The smoothing operation we describe arises in our geometric design environment in two different situations. It provides an easy method for generating solid models with curved surfaces from approximate polyhedral models that have been created interactively. The operation is also used as the last phase of solid model creation from reconstruction of medical images. Medical image reconstruction results in polyhedral models with very high feature density(vertices, edges, and faces). A density reduction step generates a rough polyhedral model from the dense model by collecting features into groups. The smoothing operation results in a low feature density model that accurately represents the medical image [7].

Generation of the surface patch is a compute intensive operation. Also, patch computation for a face is independent of that for other faces, except for continuity requirements, and can be done in parallel. However, surface curvature parameters often require interactive twiddling by the designer, in order to adhere to global or local requirements, and to control the goodness of fit. Collaborative Smoothing parallelizes this step, by allowing multiple designers concurrent access, and thus significantly improves design throughput.

4.2.9.2 Operation Outline

Smoothing is performed in two phases – curved wireframe generation and interpolating surface computation. The curved wireframe is generated in Shilp by specifying continuity parameters, as well as edge curvature and vertex normals. The curved

wireframe specifies the intersection of the interpolating surfaces, where they satisfy the specified continuity requirement (C1 or C0). Parameters for controlling normal values and edge curvature are specified graphically through the Shilp user interface. Actual fleshing of the wireframe is done by requesting service from Ganith, which is an algebraic geometry toolkit. In a single user setting, the designer specifies the parameters for all the faces, based on the requirements for the design, and subsequently computes the interpolating surfaces by making calls to Ganith servers. The obtained model is checked for goodness using still images, motion video or some calculated metric for reference, parameters are twiddled and the computation process repeated till a satisfactory model is obtained.

4.2.9.3 The Shastra setting

The Shastra environment for this operation consists of a collection of instances of the Shastra Kernel, Shilp and Ganith. A collaborative session is initiated by one of the Shilp users in the environment. He specifies, to the local Kernel, the list of Shilp users that will be invited to participate in the session, and becomes the group leader. The Kernel instantiates a Session Manager, which starts a session with the group leader as its sole participant, and then invites the specified users of concurrently executing remote Shilp sessions to participate. Users that accept are incorporated into the session. The Group Leader uses the access regulation mechanism to specify what the other participants can/can not do. He can invite other users to join the session at any point in the collaboration. Other remote users can request to join the session, and current participants can leave the session at any time.

4.2.9.4 The Collaborative Operation

Every participating Shilp session creates a shared window in which all the cooperative interaction occurs. A local window which displays only the user's sub-problem can also be created. A user introduces the object to be smoothed by selecting it into the Collaboration Window. The Session Manager is responsible for providing

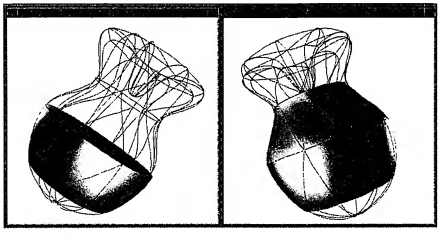

Figure 4.11 One Site in a Collaborative Smoothing Scenario in Shilp

access to the object at all participating sites that have the Access permission, and for permitting interaction relevant to the operation at sites that have Modify permission for the collaboration.

The Session Manager partitions the object into non-overlapping zones when the smoothing operation is initiated. The partitioning is based on the number of people in the collaboration, and on the number of subtasks left in the operation (the number of uninterpolated faces, in this case). It aims to minimize the number of boundary edges of partitions, which are regions of contention in this collaboration, since adjacent faces have to obey the continuity requirement along shared edges. The partitioning defines a scenario for minimal-conflict cooperative interaction. The partitioning can be dynamically altered as users join/leave the session. The group leader can explicitly specify and alter the partitioning.

The partitioned zones are assigned to the collaborating designers, and are colored differently for identification. Every user is responsible for smoothing his zone by first generating a satisfactory curved wireframe and subsequently using instances of the algebraic geometry toolkit, Ganith, to perform the actual interpolation and cycling through this process till a satisfactory design is created. Figures 4.11 and 4.12 show

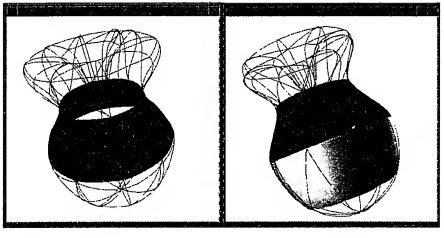

Figure 4.12 Another Site in the Collaborative Smoothing Session

a view of the interaction at two sites, with the users' own zone in a local window, and the status of the operation in the shared window.

4.2.9.5 Collaborative Interaction

Interaction can occur in two modes. In the Regulated mode, the user responsible for a zone controls the normals and curvature control parameters for vertices, edges and faces internal to the zone. All other users are denied access to the interior by the session manager. In the Unregulated mode, the partitioning merely suggests a minimal-conflict setting, and the session manager doesn't regulate interaction beyond what is specified by the permission settings for the site and the object. In this mode any user can alter any parameter if he has Access and Modify permissions for the collaboration.

The algebraic continuity requirement imposes constraints at the boundary of a partition. Users of adjacent zones must agree about parameter settings for boundary edges so that continuity requirements are not violated. A good group design protocol for this setting is to resolve boundary condition issues at the start of the operation, to prevent unnecessary cycles due to inconsistencies later on.

The session manager allows only one user to manipulate a "hot spot" – where there is a possibility of contention – at any particular instant. It uses the first-come-first-served paradigm to decide which user gets temporary exclusive control. The last validly specified parameter value takes effect. Designers can agree on a parameter adjustment protocol using the token passing facility of the system to take turns to specify vertex normals and edge curvature along boundary edges. Alternately, they can use auxiliary communication channels (audio or text) by initiating Sha-Phone or Sha-Talk sessions to decide mutually acceptable values, and which users will set those values. A Sha-Video session can be used to inspect a physical model or picture to visually establish goodness of the smoothing operation.

All operations are performed *via* the centrall session manager which is responsible for keeping all sites up-to-date, so that the users have a dynamically changing and

continuously updated view of the operation in the shared window – the curved wireframe and interpolated patches of the object. Changing a parameter requires all of its dependencies to be re-evaluated. The operation is completed when all the polyhedral faces have been smoothed. Any site with Copy permission can then extract the model from the session and save it.

4.2.9.6 Regulation Context

A point to note is that the topology of the object, as defined by the connectivity of vertices and edges, does not change in the entire operation. Thus the wireframe skeleton of the desired result serves as a context for the collaborative task and is always available to the collaborating designers – in some sense they know what the resulting object will look like, and it provides a very convenient medium to express partitioning information as well as collaborative task status information. A collaborator who joins an ongoing collaboration late can quickly come up-to-date, and infer the status of the operation.

4.2.10 Heterogeneous Collaboration

The above described application is an example of a homogeneous collaboration – the collaborative design task is supported on a collection of instances of the same tool (Shilp in this case). We are currently building an environment for collaborative design of custom hip and knee implants, using different toolkits of Shastra coupled with a computer aided manufacturing facility. This puts us in the realm of heterogeneous collaborations – here collaborations are supported between instances of different tools, which operate on different types of models or data.

The architecture paradigm of Shastra has greatly facilitated the kind of inter-application cooperation required to build such a system. The Medical Image Reconstruction toolkit (Vaidak) is used to build a model of the patients femur from cross-section information (CT/MRI images). A designer uses Shilp to custom design an implant for the femur (see Figure 4.3). A physical analyst using Bhautik conducts

a stress-strain analysis to evaluate the load transfer occurring between implant and bone. In Figure 4.4 a designer uses Bhautik to analyze stress under loading patterns to optimize custom implant shape for an artificial implant for a human femur (top-left). The designers use video-conferencing for communication. The design team iterates over this custom design process till an optimally shaped customized implant is obtained. Multimedia communication facilities in the form of Sha-Video, Sha-Phone and Sha-Poly conferences permit a rapid exchange of rationales for design choices, interpretations of analyses and iterative shape modification and analysis. This is described in detail in [14].

5. CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

We have defined high level semantic models for tool architecture, media integration and interaction, and for collaboration and sharing. We have created an infrastructure that understands the core underlying technologies of CSCW and provides abstractions that enable application developers to build groupware. Finally, we have demonstrated the viability of our models and infrastructure by building multi-user tools and collaborative problem solving environments.

5.1.1 Models

We have attempted to fill the need for high level semantic models that will enable developers to build groupware more easily. The model stresses openness and extensibility because it is unlikely that any specific software tool will ever encompass all the functionality that a user might reasonably require. The open architecture of the tool model supports extensibility by integration with independent tools. It provides cooperation *via* interoperation. It provides a highly generalized mechanism for integrating a heterogeneous range of information technologies. Interoperation allows the function and content of any tool to be accessed by another tool. Various tools can be cross coupled and linked in a variety of interactive ways.

5.1.1.1 Structural Model

The Structural Model described in Section 2.1 is an architectural model for distributed and collaborative tools that emphasizes the separation of interface and function. In this model tools consist of "contexts" that are characterized by "state" that

is modified by "events". Contexts may be local or remote. State may be private or shared. Events may be user "actions" or "triggers". Events affect the private or shared state and can cause multiple contexts, both local and remote, to be altered simultaneously, synchronously or asynchronously.

The tool is thought of as an event driven data flow machine that has mechanisms for routing events to different states and contexts. Distributed and collaborative tools are built by setting up the appropriate state and contexts, and by describing how events alter them. This model is amenable to implementing the core requirements for enabling groupware. An important point to note is that the model provides a means of structuring tool design. It makes no assumptions about language or platform of implementation. In fact, tools built using this model can freely interoperate with others built on the same principles but on different platforms, as long as they use a compatible messaging mechanism.

5.1.1.2 Media Model

The Media Model described in Section 2.2 is a formalism that enables media integration into tools. Any form of structured data with well defined interaction semantics is treated as media. In this model tools interact with media "agents" that receive input from "sources", apply "filters" to the media stream, and generate output to "sinks". In conjunction with the Structural Model, this enables multimodal user interaction, distributed interoperation, and synchronous and asynchronous conferencing.

This model enables integration of audio, video, 2D and 3D graphics, and text into tools, and extends to support application specific objects, spreadsheets, databases, animations, simulations, and hypertext and hypermedia. The recommendation that media agents be built using the Structural Model, and that they provide their functionality through the abstractions of Stub Widgets and Media Widgets enables easy integration of any media type into other tools.

5.1.1.3 Collaboration Model

The Collaboration Model is described in Section 2.3. This is a flexible model for collaboration that supports media-enhanced multi-user interaction. The sharing model extends the content and function sharing of interoperation by providing mechanisms to control and regulate synchronous and asynchronous shared interaction. The model can implement traditional centralized and replicated collaborative tools more efficiently than the state of the art. It also supports a new Session Model for collaboration. The Session Model allows for persistence and asynchronous interaction.

5.1.2 Infrastructure

The abstractions stress on semantic level handling, hiding actual details of lower level implementation. We accept and acknowledge heterogeneity in the real world, and capture and encapsulate it in the abstractions.

Of the core technological requirements of CSCW infrastructures, shared data management tends to be domain dependent, and can be implemented using many methods. We implement a method based on the simplicity of the Session Model. For concurrency control, a mature field for which well known techniques exist, we implement a simple method which, again, exploits the simplicity of the Session Model. We do not propose any new ideas in these areas. Coordination control is inherently domain and task specific, We do not attempt to specify general models and techniques, and implement it on top of the communication infrastructure. We use a flexible method for access control that meets the needs for interaction control.

We target the inadequacies of high level abstractions for distribution control, collaboration control, multimedia, graphics, and user interfaces We present an infrastructure that attempts to fill the gaps in order to support virtual spaces for flexible collaborative interaction. The infrastructure lets us build tools with shared drawing and viewing surfaces by supporting content dependent sharing – the tools are collaboration aware, and support synchronous multi-user manipulations of application-specific objects. This adds a new dimension to the kind of cooperation that can occur

in collaborative problem solving, because it permits cooperative browsing of objects and interaction in the context of tools that manipulate those objects. Since tools understand the structure of the data they manipulate, this allows a great degree of flexibility in sharing and concurrency control. It supports cooperation in the design and problem-solving phase, as well as in the review and analysis phase.

The CSCW infrastructure of Shastra facilitates creation of collaborative multimedia applications. Shastra provides intuitive session initiation methods, flexible interaction modes, and dynamic access regulation.

5.1.2.1 Distribution Substrate

The Distribution Substrate is described in Section 3.2.1. This fulfills the need for distribution control, and provides a mechanism to implement shared data management for CSCW. It enables client-server and peer-peer interaction. The substrate provides mechanisms of setting up connections across the network, and flexibly managing data in a distributed setting. It provides device independent data transport for heterogeneous environments. It implements synchronous and asynchronous remote procedure calling and provides multiple-connection management between instances of tools. It supports several application level communication protocols.

5.1.2.2 Collaboration Substrate

The Collaboration Substrate is described in Section 3.2.2. This fulfills the need for Collaboration control and provides mechanism for interaction control and access regulation. It enables multi-user interaction. The substrate uses the distribution substrate to implement shared state and context in a distributed setting. It provides session management, interaction control and access regulation facilities that enable rapid prototyping and development of collaborative tools and groupware.

5.1.2.3 Portable Graphics

The Portable Graphics Substrate is described in Section 3.2.3. This is an abstract 3D graphics system that enables dealing with 3D graphical interaction at a semantic level. It lets tools access hardware graphics facilities of workstations in a device-independent manner, by presenting a high level interface to 3D graphics. It provides source code level compatibility across different graphics platforms in a heterogeneous setting, by implementing a hardware independent graphics library.

5.1.2.4 Collaborative Graphics Substrate

The Collaborative Graphics Substrate is described in Section 3.2.4. It is based on the Structural Model and uses the distribution, collaboration and graphics substrates to implement device independent distributed and collaborative graphics. It supports synchronous and asynchronous 2D and 3D graphical interaction in a heterogeneous setting. It enables incorporation of graphics facilities into tools. It provides high level control of display and visualization parameters and supports telepointing.

5.1.2.5 Portable Multimedia

The Portable Multimedia Substrate is described in Section 3.2.5. This abstract multimedia system provides access to available hardware audio and video facilities on a workstation in a device-independent manner. It enables semantic handling of audio and video streams and interaction. It provides source code level compatibility across multiple platforms. It encapsulates details of media format and device specific interaction, providing a high level abstraction for development of multimedia tools. It deals with the issue of heterogeneity for CSCW.

5.1.2.6 Collaborative Multimedia Substrate

The Collaborative Multimedia Substrate is described in Section 3.2.6. It is based on the Structural Model and uses the distribution, collaboration and multimedia

substrates to implement device independent distributed and collaborative multimedia. It enables incorporating multimedia features and facilities into tools, and supports collaborative multimedia interaction.

5.1.3 Tools

We have described applications that demonstrate that collaboration in the scientific design setting is facilitated by multimedia support as well as information exchange.

5.1.4 Collaborative Tools

Sha-Draw and Sha-Poly (described in Appendix A) are collaborative graphics tools. Sha-Phone, Sha-Video, and Sha-Talk are multimedia conferencing tools. They are described in Appendix B. Sha-Chess (described in Appendix D) is the implementation of a virtual chess board that supports synchronous multi-user interaction in a distributed setting. Shilp is a solid modeling toolkit that supports synchronous participatory collaborative design. It is described in Appendix C.

5.1.5 Collaborative Applications

We have described Shastra, a collaborative multimedia environment, and some problem solving scenarios in Section 4. The environment for collaborative geometric design is also described in [8, 10]. The environment for collaborative custom design of artificial implants for human limbs is described in [14]. A distributed and collaborative volume visualization environment is described in [15].

Shastra is a distributed and collaborative toolkit prototyping environment. It provides a substrate for design of collaborative systems. The multimedia aspect brings communication primitives to the desktop. The integration of 3D graphics into the environment adds a new dimension to the potency of this environment, as visual processing on powerful graphics engines becomes more common. Collaboration support in the environment, in the form of communication facilities and application

development substrate, makes it easy to develop synchronous multi-user applications, and problem solving tools.

5.2 Applications

In the scientific domain, Carlborn *et al* [29] present a teleconferencing approach to modeling and analysis of empirical data, and discuss a collaborative scientific visualization environment, with output images of visualizations shared among multiple users. The Shastra environment makes it convenient to build collaborative visualization facilities that not only share results of visualizations but also the input data and models. This sharing allows multiple users to interact over the data set while analyzing multiple simultaneous renderings with different viewing directions, cutaways and independent visualization parameters. Mercurio *et al* [91] describe an interactive visualization environment for 3D imaging where an electron microscope is used as a computer peripheral. The Shastra layer promotes sharing of such unusual and expensive resources among multiple users across a network by enabling application level cooperation.

Though a scientific manipulation environment has been the focus of our implementation, the facilities easily abstract out to a variety of situations requiring similar substrates. The collaborative layer is generic and can be used to implement the heart of systems for collaborative editing, code viewing and quality assurance tools, software development environments, multi-user electronics CAD, architecture CAD and mechanical CAD tools, and interactive multi-player games etc.

5.3 Future Work

The Shastra infrastructure provides us with powerful prototyping facilities for sophisticated distributed multimedia applications and groupware. We briefly discuss some research issues and applications of this substrate.

5.3.1 Language Based Generation

An interesting issue is that of automatic generation of groupware from appropriately structured single-user tools based on a high level description language. The language would provide mechanisms to capture and express the elements of user interaction with the application, and generate a multiuser version based on the Shastra infrastructure. This would automate and further ease the task of groupware creation.

5.3.2 Collaborative Hypermedia

The Shastra infrastructure can be used to build environments for collaborative hypermedia browsing. As opposed to shared visual surfaces that existing systems allow us to build, Shastra enables shared application interfaces and provides facilities to let the user interact with and manipulate reviewable shared material. This entails developing a general formalism that captures navigation through webs of information for both private and shared interaction.

5.3.3 Shared Visual Programming

The Shastra infrastructure can be used to build visual programming and direct manipulation interfaces to systems. This can be used to implement concept maps and semantic networks, and to control animations and simulations.

5.3.4 Multimodal Interaction

The Shastra models emphasize and build on the separation of interface and function. Shastra tools are essentially interpreters of embedded command languages that respond to commands from multiple interfaces. Any system or tool that can generate expressions in the embedded command language can thus drive these tools. This eases integration of non-conventional input and output devices like touch-screens, graphics tablets, 3D mouse etc. The infrastructure can be used in conjunction with speech

recognition systems to build voice or audio-cue driven tools (*cf.* [120]). It can be used in conjunction with image-processing systems to build visual-cue driven systems.

5.3.5 Virtual Environments

The collaborative multimedia and graphics facilities of Shastra enable the creation of virtual environments. Many domain-specific applications can be implemented on top of such environments. The communication facilities can be harnessed to provide collaborative navigation through these virtual worlds, along with facilities to express remote presence, and to interact with it.

5.3.6 Implementation Issues

The current implementation of the Shastra infrastructure needs testing and enhancements that would make it a richer substrate.

5.3.6.1 Testing

Groupware adoption is very sensitive to ergonomic issues, and design and deployment needs to be a participatory process involving user feedback. We need to conduct more usability and performance tests, and incorporate the results into the control and interaction policies that have been implemented.

5.3.6.2 Constraint Management

We currently implement constraint management in an *ad hoc* manner. Though it is sufficient for the applications we have implemented, the infrastructure would benefit from the use of more formal techniques, in the interest of generality. We are investigating language based constraint specification and resolution systems (*cf.* [39]).

5.3.6.3 Concurrency Control

The Session Model for collaboration centralizes shared activity in the Session Manager and greatly eases the task of concurrency control, since it simplifies serialization. The current implementation assumes that the Session Model will be used. We need to implement a more general and powerful technique that is appropriate for other models of collaboration.

5.3.6.4 Access Control

Though the current access control mechanism works well for low level session and interaction control, we need to use a more formal, inheritance based model for general specification and regulation of access control of application objects [121].

5.3.6.5 Distribution Platform

The current system is implemented on Unix platforms and uses a custom distributed system built on top of TCP/IP. A DCE [115] based distributed implementation, that would make the system more portable, is planned.

5.3.6.6 Language Support

The Structural Model makes no assumptions about the language and platform of tool implementation, as long as it is compatible with the underlying messaging system. The current system is implemented in C. Bindings to other languages are planned.

5.3.6.7 Persistence

We need to complete implementing persistence of collaborative sessions. This will allow session state to be saved to stable store, transported, and subsequently restored, permitting asynchronous cooperative interaction.

5.4 Shastra

Shastra supports the paradigm shift in Computer Supported Cooperative Work that has enabled users to be aware of other users of systems and tools, and allows interaction among the users. This has extended the notion of sharing beyond the simple sharing of data to the sharing of computation. We have attempted to aid the development and deployment of groupware by providing general models and enabling infrastructures. Groupware developers using the Shastra substrate do not have to deal with the difficult task of marrying the multiple underlying technologies in a heterogeneous distributed setting. They use the high level semantic models implemented in the infrastructure to relate them.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] H. Abdel-Wahab and M. Feit, *XTV: A Framework for Sharing X Windows Clients in Remote Synchronous Collaboration*, Proc. IEEE Conference on Communications Software (1991), 159–167.

[2] M. Abel, *Experiences in an Exploratory Organization*, Intellectual Teamwork: Social and Technological Foundations of Cooperative Work, Lawrence Erlbaum Associates (1990), 489–510.

[3] J. Adam and D. Tennenhouse, *The Vidboard: A Video Capture and Processing Peripheral for a Distributed Multimedia System*, Proc. ACM Conference on Multimedia (1993), 113–120.

[4] S. Ahuja, J. Ensor, and D. Horn, *The Rapport Multimedia Conferencing System*, Proc. ACM Conference on Office Information Systems (1988), 1–8.

[5] S. Ahuja, J. Ensor, and S. Lucco, *A Comparison of Applications Sharing Mechanisms in Realtime Desktop Conferencing Systems*, Proc. ACM Conference on Office Information Systems (1990), 238–248.

[6] M. Altenhofen, J. Dittrich, R. Hammerschmidt, T. Kappner, C. Kruschel, A. Kuckes, and T. Steinig, *The BERKOM Multimedia Collaboration Service*, Proc. ACM Conference on Multimedia (1993), 457–463.

[7] V. Anupam and C. Bajaj, *The Shilp Solid Modeling and Display Toolkit, v1.1 – A User's Guide*, Tech. Report CSD-TR-92-072, Purdue University, 1992.

[8] ———, *Collaborative Multimedia Scientific Design in Shastra*, Proc. ACM Conference on Multimedia (1993), 447–456.

[9] ———, *Shastra: An Architecture for Development of Collaborative Applications*, Proc. Second Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (1993), 155–166.

[10] ———, *Collaborative Multimedia in Scientific Design*, IEEE Multimedia Journal (1994), 39–49.

[11] ———, *Shastra: An Architecture for Development of Collaborative Applications*, International Journal of Intelligent and Cooperative Information Systems (IJICIS) (1994), in press.

[12] V. Anupam, C. Bajaj, A. Burnett, S. Cutchin, M. Fields, A. Royappa, and D. Schikore, *XS: A Hardware Independent Graphics and Windowing Library*, Tech. Report CSD-TR-91-062, Purdue University, 1991.

[13] V. Anupam, C. Bajaj, S. Cutchin, S. Evans, I. Ihm, J. Chen, A. Royappa, D. Schikore, and G. Xu, *Scientific Problem Solving in a Distributed and Collaborative Geometric Environment*, Journal of Mathematics and Computers in Simulation (1994), in press.

[14] V. Anupam, C. Bajaj, and D. Schikore, *Custom Prosthesis Design and Prototyping*, Multimedia Medical Education, Roy Rada ed., Intellect Books, U.K. (1994), in press.

[15] V. Anupam, C. Bajaj, D. Schikore, and M. Schikore, *Distributed and Collaborative Volume Visualization*, IEEE Computer (1994), in press.

[16] A. Babadi, *COMIX: A Tool to Share X Applications*, Proc. Second Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (1993), 192–196.

[17] C. Bajaj and I. Ihm, *Algebraic Surface Design with Hermite Interpolation*, ACM Transactions on Graphics **11** (1992), no. 1, 61–91.

[18] C. Bajaj and A. Royappa, *The GANITH Algebraic Geometry Toolkit*, Proc. First International Symposium on the Design and Implementation of Symbolic Computation Systems (1990), no. 429, 268–269.

[19] L. Bannon and K. Schmidt, *CSCW: Four Characters in Search of a Context*, Studies in Computer-Supported Cooperative Work: Theory, Practice and Design, J. M. Bowers and S. D. Benford eds., Proc. First European Conference on Computer-Supported Cooperative Work (1991), 3–16.

[20] T. Berners-Lee and R. Cailliau, *World-Wide Web*, Proc. Conference on Computing In High Energy Physics (1992), 23–27.

[21] N. Borenstein, *Computational Mail as Network Infrastructure for Computer-Supported Cooperative Work*, Proc. ACM Conference on Computer-Supported Cooperative Work (1992), 67–74.

[22] N. Borenstein and N. Freed, *Multipurpose Internet Mail Extensions: Mechanism for Specifying and Describing the Format of Internet Message Bodies*, Tech. Report Internet RFC 1341, Network Information Center, 1992.

[23] N. Borenstein and C. Thyberg, *Power, Ease of Use and Cooperative Work in a Practical Multimedia Message System*, International Journal of Man-Machine Studies **34** (1991), no. 2, 229–259.

[24] T. Brinck and L. M. Gomez, *A Collaborative Medium for the Support of Conversational Props*, Proc. ACM Conference on Computer-Supported Cooperative Work (1992), 171–178.

[25] D. Brittan, *Being There: The Promise of Multimedia Communications*, MIT Technology Review **95** (1992), no. 4, 42–50.

[26] L. Brothers, V. Sembugamoorthy, and M. Muller, *ICICLE: Groupware For Code Inspection*, Proc. Conference on Computer-Supported Cooperative Work (1990), 169–181.

[27] C. Bullen and J. Bennett, *Groupware in Practice: An Interpretation of Work Experiences*, Computerization and Controversy: Value Conflicts and Social Choices, C. Dunlop and R. Kling eds., Academic Press (1991), 257–287.

[28] W. Buxton, *Proc. Graphics Interface* , Morgan Kaufmann, 1992.

[29] I. Carlbom, W. Hsu, G. Klinkner, R. Szeliski, K. Waters, M. Doyle, J. Gettys, K. Harris, T. Levergood, R. Palmer, M. Picart, D. Terzopoulos, D. Tonnesen, M. Vannier, and G. Wallace, *Modeling and Analysis of Empirical Data in Collaborative Environments*, Comm. of the ACM **41** (1992), no. 6, 73–84.

[30] V. Cerf, *Networks*, Scientific American **265** (1991), no. 3, 72–81.

[31] D. Chamberlin and C. Goldfarb, *Graphic Applications of the Standard Generalized Markup Language (SGML)*, Computer Graphics **11** (1987), no. 4, 54–63.

[32] A. Clement, *Cooperative Support for Computer Work: A Social Perspective on the Empowering of End Users*, Proc. Conference on Computer-Supported Cooperative Work, F. Halasz ed., ACM (1990), 223–236.

[33] D. Comer, *Internetworking with TCP/IP*, Prentice-Hall, 1987.

[34] E. Conklin, *Capturing Organizational Memory*, Proc. Groupware, Morgan Kaufmann (1992), 133–137.

[35] J. Conklin, *Hypertext: An Introduction and Survey*, IEEE Computer **20** (1987), no. 9, 17–41.

[36] J. Conklin and M. Begeman, *gIBIS: A hypertext tool for exploratory policy discussion*, Proc. Conference on Computer-Supported Cooperative Work (1988), 140–152.

[37] E. Craighill, R. Lang, M. Schlager, and J. Garcia-Luna, *Environments to Enable Informal Collaborative Design Processes*, Proc. First Workshop on Enabling Technologies for Concurrent Engineering **1** (1992), 32–42.

[38] T. Crowley, P. Milazzo, E. Baker, H. Forsdick, and R. Tomlinson, *MMConf: An Infrastructure for Building Shared Multimedia Applications*, Proc. Conference on Computer-Supported Cooperative Work, F. Halasz ed., ACM (1990), 329–342.

[39] M. Cutkosky, M. Genesereth, R. Englemore, R. Fikes, T. Gruber, W. Mark, J. Tenenbaum, and J. Weber, *PACT: An Experiment in Integrating Concurrent Engineering Systems*, IEEE Computer **26** (1993), no. 1, 28–37.

[40] P. Dewan and R. Choudhary, *Primitives for Programming Multi-User Interfaces*, Proc. ACM Symposium on User Interface Software and Technology (1991), 41–48.

[41] P. Dourish and V. Belloti, *Awareness and Coordination in Shared Workspaces*, Proc. ACM Conference on Computer-Supported Cooperative Work (1992), 107–114.

[42] P. Dourish and S. Bly, *Portholes: Supporting Awareness in a Distributed Work Group*, Proc. ACM Conference on Computer Human Interaction (1992), 541–547.

[43] S. Dubs and S. Hayne, *Distributed Facilitation: A Concept Whose Time Has Come?*, Proc. ACM Conference on Computer-Supported Cooperative Work (1992), 314–321.

[44] C. Ellis, S. Gibbs, and G. Rein, *Groupware: Some Issues and Experiences*, Comm. of the ACM **34** (1991), no. 1, 38–58.

[45] S. Elrod, R. Bruce, R. Gold, F. Halasz, W. Janssen, D. Lee, K. McCall, E. Pederson, K. Pier, J. Tang, and B. Welch, *Liveboard: A Large Interactive Display Supporting Group Meetings, Presentations and Remote Collaboration*, Proc. ACM Conference on Computer Human Interaction (1992), 599–607.

[46] J. Eveland and T. Bikson, *Work Group Structures and Computer Support: A Field Experiment*, ACM Transactions on Office Information Systems **6** (1988), no. 4, 344–379.

[47] T. Finholt and L. Sproull, *Electronic Groups at Work*, Organization Science **1** (1990), no. 1, 41–64.

[48] R. Fish, R. Kraut, M. Leland, and M. Cohen, *Quilt - A Collaborative Tool for Cooperative Writing*, Proc. ACM Conference on Office Information Systems (1988), 85–113.

[49] N. Flor and E. Hutchins, *Analyzing Distributed Cognition in Software Teams: A Case Study of Team Programming During Perfective Software Maintenance*, Proc. Fourth Workshop on Empirical Studies of Programmers, Ablex (1991), 36–64.

[50] F. Flores, M. Graves, B. Hartfiels, and T. Winograd, *Computer Systems and the Design of Organizational Interaction*, ACM Transactions on Office Information Systems **6** (1988), no. 2, 153–172.

[51] E. Fox, *Advances in Interactive Digital Multimedia Systems*, IEEE Computer **24** (1991), no. 10, 9–21.

[52] B. Gaines and M. Shaw, *Open Architecture Multimedia Documents*, Proc. ACM Conference on Multimedia (1993), 137–146.

[53] D. Garfinkel, P. Gust, M. Lemon, and S. Lowder, *The SharedX Multi-user Interface User's Guide, Version 2.0*, Tech. Report STL-TM-89-07, Hewlett Packard Laboratories, Palo Alto, California, 1989.

[54] W. Gaver, *Sound Support for Collaboration*, Readings in Groupware and Computer-Supported Cooperative Work: Assisting Human-Human Collaboration, Ronald M. Baecker ed., Morgan Kaufman (1993), 355–362.

[55] S. Gibbs, *LIZA : An Extensible Groupware Toolkit*, Tech. Report Report STP-042-88, MCC Software Technology Program, 1988.

[56] ———, *Composite Multimedia and Active Objects*, Proc. Conference on Object Oriented Programming Systems, Languages and Applications (1991), 97–112.

[57] Y. Goldberg, M. Safran, and E. Shapiro, *Active Mail - A Framework for Implementing Groupware*, Proc. ACM Conference on Computer-Supported Cooperative Work (1992), 75–83.

[58] T. Graham and T. Urnes, *Relational Views as a Model for Automatic Distributed Implementation of Multi-User Applications*, Proc. ACM Conference on Computer-Supported Cooperative Work (1992), 59–66.

[59] S. Greenberg, *Sharing Views and Interactions with Single-User Applications*, Proc. ACM Conference on Office Information Systems (1990), 227–237.

[60] S. Greenberg, M. Roseman, D. Webster, and R. Bohnet, *Issues and Experiences Designing and Implementing Two Group Drawing Tools*, Proc. Twenty-Fifth Annual Hawii International Conference on the System Sciences **4** (1992), 139–150.

[61] I. Greif, *Designing Group-enabled Applications: A Spreadsheet example*, Proc. Groupware, D. Coleman ed., Morgan Kaufmann (1992), 515–525.

[62] J. Grudin, *Groupware and Cooperative Work: Problems and Prospects*, The Art of Human-Computer Interface Design, B. Laurel ed., Addison-Wesley (1990), 171–185.

[63] N. Guimaraes, P. Silva, J. Santos, and A. Siemaszko, *MObViews: A Multiuser Worksheet for a Mechanical Enginering Environment*, Proc. Second Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (1993), 182–186.

[64] J. Haake and B. Wilson, *Supporting Collaborative Writing Of Hyperdocuments in SEPIA*, Proc. ACM Conference on Computer-Supported Cooperative Work (1992), 138–146.

[65] S. Harrison and S. Minneman, *The Media Space: A Research Project into the use of Video as a Design Medium*, Proc. Conference on Participatory Design (1990), 43–58.

[66] C. Heath and P. Luff, *Disembodied Conduct: Communication Through Video in a Multi-Media Office Environment*, Proc. ACM Conference on Computer Human Interaction (1991), 99–103.

[67] D. Heller and P. Ferguson, *Motif Programming Manual*, vol. 6A, O'Reilly & Associates Inc., December 1993.

[68] R. Hill, *Languages for Construction of Multi-User, Multi-Media Synchronous (MUMMS) Applications*, Languages for Developing User Interfaces, Brad Meyers ed., Jones and Bartlett (1992), 125–143.

[69] J. Hollan and S. Stornetta, *Beyond Being There*, Proc. ACM Conference on Computer Human Interaction (1992), 119–125.

[70] H. Ishi and M. Kobayashi, *Clearboard: A Seamless Medium for Shared Drawing and Conversation with Eye Contact*, Proc. ACM Conference on Computer Human Interaction (1992), 525–532.

[71] H. Ishi and M. Ohkubo, *Design of TeamWorkStation: A Realtime Shared Workspace Fusing Desktops and Computer Screens*, Multi-User Interfaces and Applications, S. Gibbs and A. Verrijn-Stuart eds., North Holland (1990), 131–142.

[72] K. Jeffay, J. Lin, J. Menges, F. Smith, and J. Smith, *Architecture of the Artifact-Based Collaboration Matrix*, Proc. ACM Conference on Computer-Supported Cooperative Work (1992), 195–202.

[73] R. Johansen, *Computer Support for Business Teams*, The Free Press, 1988.

[74] S. Kaplan, W. Tolone, D. Bogia, and C. Bignoli, *Flexible, Active Support for Collaborative Work with ConversationBuilder*, Proc. ACM Conference on Computer-Supported Cooperative Work (1992), 378–385.

[75] R. Keller and W. Effelsberg, *MCAM: An Application Layer Protocol for Movie Control, Access and Management*, Proc. ACM Conference on Multimedia (1993), 21–28.

[76] R. Kraut, R. Fish, R. Root, and B. Chalfonte, *Informal Communication in Organizations: Form, Function, and Technology*, People's Reactions to Technology in Factories, Offices and Aerospace, S. Oskampand and S. Spacapan eds., Sage Publications (1990), 145–199.

[77] M. Krueger, *Artificial Reality*, Addison-Wesley, 1991.

[78] K. Lai, T. Malone, and K. Yu, *Object Lens: A "Spreadsheet" for Cooperative Work*, ACM Transactions on Office Information Systems **6** (1988), no. 4, 332–396.

[79] G. Landow, *Hypertext and Collaborative Work: The Example of Intermedia*, Intellectual Teamwork: Social and Technological Foundations of Cooperative Work, J. Galegher, R. Kraut and C. Egido eds., Lawrence Erlbaum Associates (1990), 407–428.

[80] B. Lange, *Electronic Group Calendaring*, Proc. Groupware, D. Coleman ed., Morgan Kaufmann (1992), 428–432.

[81] J. Lauwers, T. Joseph, K. Lantz, and A. Romanov, *Replicated Architectures for Shared Window Systems: A Critique*, Proc. ACM Conference on Office Information Systems (1990), 249–260.

[82] J. Lauwers and K. Lantz, *Collaboration Awareness in Support of Collaboration Transparency: Requirements for the Next Generation of Shared Window Systems*, Proc. ACM Conference on Computer Human Interaction (1990), 303–311.

[83] J. Lee, *XSketch: A Multi-User Sketching Tool for X11*, Proc. ACM Conference on Office Information Systems (1990), 169–173.

[84] T. Levergood, A. Payne, J. Gettys, G. Treese, and L. Stewart, *AudioFile: A Network-Transparent System for Distributed Audio Applications*, Proc. Usenix Summer Conference (1993), 22–33.

[85] S. Lu, K. Smith, A. Herman, D. Mattox, M. Silliman, M. Lucenti, J. Jacobs, D. Chazin, M. Lawley, and M. Case, *SWIFT: Software Workbench for Integrating and Facilitating Teams*, Proc. Second Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (1993), 48–59.

[86] T. Malone, , K. Grant, K. Lai, R. Rao, and D. Rosenblitt, *The Information Lens: An Intelligent System for Information Sharing and Coordination*, Technological Support for Work Group Collaboration, M. H. Olson ed., Lawrence Erlbaum Associates (1989), 65–88.

[87] T. Malone and K. Crowston, *What is Coordination Theory and How Can it Help Design Cooperative Work Systems.*, Proc. Conference on Computer-Supported Cooperative Work, F. Halasz ed., ACM (1990), 357–370.

[88] T. Malone, K. Lai, and C. Fry, *Experiments with Oval: A Radically Tailorable tool for Cooperative Work*, Proc. ACM Conference on Computer-Supported Cooperative Work (1992), 289–297.

[89] M. Mantei, *Observation of Executives Using a Computerized Supported Meeting Environment.*, International Journal of Decision Support Systems **5** (1989), 153–166.

[90] ———, *Adoption Patterns for Media Space Technology in a University Research Environment*, Friend '21 Conference (1991), 1–8.

[91] P. Mercurio, T. Elvine, S. Young, P. Cohen, K. Fall, and M. Ellisman, *The Distributed Laboratory*, Comm. of the ACM **41** (1992), no. 6, 54–63.

[92] V. Mey and S. Gibbs, *A Multimedia Component Kit: Experiences with Visual Composition of Applications*, Proc. ACM Conference on Multimedia (1993), 291–300.

[93] K. Narayanaswamy and N. Goldman, *"Lazy" Consistency: A Basis for Cooperative Software Development*, Proc. ACM Conference on Computer-Supported Cooperative Work (1992), 257–264.

[94] B. Nardi and J. Miller, *Twinkling lights and Nested Loops: Distributed Problem Solving and Spreadsheet Development*, Computer-Supported Cooperative Work and Groupware, S. Greenberg ed., Academic Press (1991), 29–52.

[95] B. Naylor and W. Thibault, *Set Operations on Polyhedra using Binary Space Partitioning Trees*, IEEE Computer Graphics **21** (1987), no. 4, 65–77.

[96] C. Neuwirth, D. Kaufer, R. Chandhok, and J. Morris, *Issues in the Design of Computer Support for Co-Authoring and Commenting*, Proc. Conference on Computer-Supported Cooperative Work, F. Halasz ed., ACM (1990), 183–195.

[97] S. Newcomb, N. Kipp, and V. Newcomb, *Hytime: Hypermedia/Time-based Document Structuring Language*, Comm. of the ACM (1991), 67–83.

[98] R. Newman-Wolfe, M. Webb, and M. Montes, *Implicit Locking in the Ensemble Concurrent Object-Oriented Graphics Editor*, Proc. ACM Conference on Computer-Supported Cooperative Work (1992), 265–272.

[99] J. Nunamaker, A. Dennis, J. Valacich, D. Vogel, and J. George, *Electronic Meeting Systems to Support Group Work.*, Comm. of the ACM **34** (1991), no. 7, 40–61.

[100] J. Nunamaker, D. Vogel, A. Heminger, B. Martz, R. Grohowski, and C. McGoff, *Experiences at IBM with Group Support Systems: A Field Study*, Decision Support Systems **5** (1989), no. 2, 183–196.

[101] A. Nye, *Xlib Programming Manual*, vol. 1, O'Reilly & Associates Inc., February 1992.

[102] A. Nye and T. O'Reilly, *X Toolkit Intrinsics Programming Manual*, vol. 4M, O'Reilly & Associates Inc., August 1992.

[103] OMG, *The Common Object Request Broker: Architecture and Specification*, Tech. Report OMG Document #91.12.1, Object Management Group, December 1991.

[104] J. Patterson, R. Hill, S. Rohall, and M. Meeks, *Rendezvous: An Architecture for Synchronous Multi-User Applications*, Proc. ACM Conference on Computer-Supported Cooperative Work (1990), 317–328.

[105] A. Pinsonneault and K. Kraemer, *The Impact of Technological Support on Groups: An Assesment of Empirical Research*, Decision Support Systems **5** (1989), no. 2, 197–216.

[106] I. Posner and R. Baecker, *How People Write Together*, Proc. Twenty-Fifth Annual Hawaii International Conference on the System Sciences **4** (1992), 127–138.

[107] A. Prakash and M. Knister, *Undoing Actions in Collaborative Work*, Proc. ACM Conference on Computer-Supported Cooperative Work (1992), 273–280.

[108] R. Price, *MHEG: An Introduction to the Future International Standard for Hypermedia Object Interchange*, Proc. ACM Conference on Multimedia (1993), 121–128.

[109] V. Rangan and H. Vin, *System Support for Computer Mediated Multimedia Collaborations*, Proc. ACM Conference on Computer-Supported Cooperative Work (1992), 203–209.

[110] B. Reeves and F. Shipman, *Supporting Communication between Designers with Artifact-Centered Evolving Information Spaces*, Proc. ACM Conference on Computer-Supported Cooperative Work (1992), 394–401.

[111] R. Rice and C. Steinfield, *Experiences with New Forms of Organizational Communication via Electronic Mail and Voice Messaging*, Telematics and Work, J. Andriessen and R. Roe eds., Lawrence Erlbaum Associates (1991), 32–45.

[112] M. Robinson, *Computer-Supported Cooperative Work: Cases and Concepts*, Groupware 91: The Potential of Team and Organisational Computing, P. Hendriks ed., SERC (1991), 59–75.

[113] T. Rodden and G. Blair, *CSCW and Distributed Systems: The Problem of Control.*, Proc. Second European Conference on Computer-Supported Cooperative Work, L. Bannon, M. Robinson and K. Schmidt eds., Kluwer Academic Publishers (1991), 49–64.

[114] M. Roseman and S. Greenberg, *A Groupware Toolkit for Building Real-Time Conferencing Applications*, Proc. ACM Conference on Computer-Supported Cooperative Work (1992), 43–50.

[115] W. Rosenberry, D. Kenney, and G. Fisher, *Understanding DCE*, O'Reilly & Associates, Inc., 1993.

[116] S. Sarin and I. Greif, *Computer-based Real-time Conferencing Systems*, IEEE Computer **18** (1985), no. 10, 33–45.

[117] M. Scardamalia and C. Bereiter, *High Levels of Agency for Children in Knowledge Building: A Challenge for the Design of New Knowledge Media*, The Journal of the Learning Sciences **1** (1991), no. 1, 37–68.

[118] B. Schatz, *Building an Electronic Scientific Community.*, Proc. Twenty-Fifth Annual Hawaii International Conference on the System Sciences (1991), 739–748.

[119] R. Scheifler, J. Gettys, and R. Newman, *The X Window System*, ACM Transactions on Graphics **5** (1986), no. 2, 79–109.

[120] C. Schmandt, *Phoneshell: the Telephone as Computer Terminal*, Proc. ACM Conference on Multimedia (1993), 373–382.

[121] H. Shen and P. Dewan, *Access Control for Collaborative Environments*, Proc. ACM Conference on Computer-Supported Cooperative Work (1992), 51–58.

[122] L. Shu and W. Flowers, *Groupware Experiences in Three Dimensional Computer Aided Design*, Proc. ACM Conference on Computer-Supported Cooperative Work (1992), 179–186.

[123] L. Sproull and S. Kiesler, *Connections: New Ways of Working in the Networked Organization*, The MIT Press, 1991.

[124] R. Sproull, *A Lesson in Electronic Mail.*, Connections: New Ways of Working in the Networked Organization, L. Sproull and S. Kiesler eds., The MIT Press (1991), 177–184.

[125] K. Srinivas, R. Reddy, A. Babadi, S. Kamana, V. Kumar, and Z. Dai, *MONET: A Multi-media System for Conferencing and Application Sharing in Distributed Systems*, Proc. First Workshop on Enabling Technologies for Concurrent Engineering **1** (1992), 21–37.

[126] D. Sriram, R. Logcher, A. Wong, and S. Ahmed, *An object-oriented framework for collaborative engineering design*, Computer Aided Product Development, MIT-JSME Workshop, MIT (1991), 51–92.

[127] M. Stefik, D. Bobrow, G. Foster, S. Lanning, and D. Tatar, *WYSIWIS Revised: Early Experiences with Multiuser Interfaces*, ACM Transactions on Office Information Systems **5** (1987), no. 2, 147–167.

[128] J. Tang, *Findings from Observational Studies of Collaborative Work*, International Journal of Man-Machine Studies **34** (1991), no. 2, 143–160.

[129] D. Tatar, G. Foster, and D. Bobrow, *Design for Conversation: Lessons from Cognotor*, International Journal of Man-Machine Studies **34** (1991), no. 2, 185–209.

[130] G. Toye, M. Cutkosky, L. Leifer, J. Tenenbaum, and J. Glicksman, *SHARE: A Methodology and Environment for Collaborative Product Development*, Proc. Second Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (1993), 33–47.

[131] M. Turoff, *Computer Mediated Communication Requirements for Group Support*, Journal of Organizational Computing **1** (1991), 85–113.

[132] K. Watabe, S. Sakata, K. Maeno, H. Fukuoka, and T. Ohmori, *A Distributed Multiparty Desktop Conferencing System*, Proc. ACM Conference on Computer-Supported Cooperative Work (1990), 27–38.

[133] S. Whittaker, S. Brennan, and H. Clark, *Coordinating Activity: An analysis of interaction in CSCW*, Proc. Conference on Computer Human Interaction: Human Factors in Computing Systems **1** (1991), 441–442.

[134] T. Winograd, *Groupware and the Emergence of Business Technology*, Proc. Groupware, D. Coleman ed., Morgan Kaufmann (1992), 69–72.

[135] C. Wolf, J. Rhyne, and L. Briggs, *Communication and Information Retrieval with a Pen-based Meeting Support Tool*, Proc. ACM Conference on Computer-Supported Cooperative Work (1992), 322–329.

APPENDICES

Appendix A: Graphics Support

A.1 Sha-Draw

Sha-Draw is an Agent in the Shastra environment, built on the Structural and Media Models. (The notion of Agents, and these Models, is described in Section 2.) Sha-Draw uses structured 2D graphics and interaction as its media type.

Sha-Draw is used as a sketching tool, and facilitates the generation and display of simple 2D sketches and pictures, using a rich set of primitives. The typical user interface is shown in Figure A.1. It depicts the drawing toolbox (at left) that is used to choose drawing primitives, and a drawing canvas (at right) that shows some primitives drawn. Also shown are a few pointers or markers, which are typically used to point to features of a drawing.

Sha-Draw allows the user to create and use multiple canvases (Contexts in the Model). Drawings can be moved to and from different canvases using interface facilities. Sha-Draw supports input and output of 2D drawings from files. They are the data objects that it manipulates. Figure A.2 depicts the block architecture of Sha-Draw.

In Sha-Draw, the actual interactive process of creation of drawings is captured using an embedded command language. This interaction contains data with temporal attributes and constitutes a media stream. Sources for this media stream can be the actual canvas, local files, or remote sources like other tools. Sinks for this media stream can be the local canvas, local files, or remote sinks like other tools. The embedded command language can be used as a scripting language for simple animations using 2D graphics primitives for drawing. Filters allow the application of 2D transformations, color changes and stream mixing.

Drawings are recorded into files by setting up the canvas as a Source, and the file as a Sink. They are played back from files by setting up the file as a Source, and the canvas as a Sink. The interface provides transport control facilities.

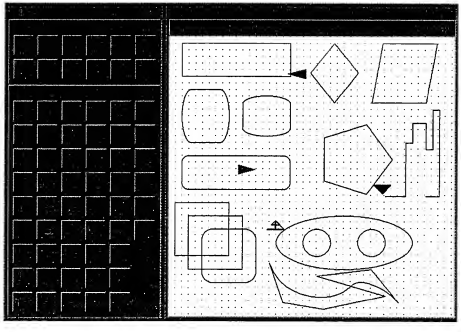

Figure A.1 Sha-Draw User Interface

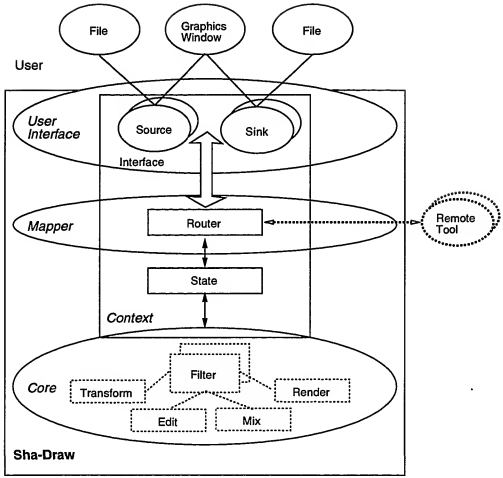

Figure A.2 High Level Architecture of Sha-Draw

In the distributed setting, Sha-Draw allows a user to draw into canvases of other Sha-Draw instances running on different machines across the network. This is done by setting up the local canvas as a Source, and the remote canvases as Sinks. Alternately, the user plays back recorded files into multiple remote canvases by setting up the local file as a Source, and the local and remote canvases as Sinks. Only the user controlling the Source can draw into the canvas, though everyone sees the interaction or the drawing. The telepointing facility is exploited for gesturing and pointing.

In the collaborative setting, Sha-Draw is used as a multi-user 2D graphics system and sketchpad. A collaborative session consisting of Sha-Draw instances lets a group of collaborators synchronously create and edit 2D sketches on shared virtual whiteboards. When a user joins the session, Sha-Draw creates a shared canvas. Drawing and interaction streams from all sites are mixed and rendered into this canvas. In the simple implementation, only graphical objects that are drawn into this canvas are shared with every other user in the session, by transporting the appropriate data object. In the more complex case, the interaction involved in drawing is shared, by redirecting the input media stream to the shared Session Context, i.e. the new canvas is the Source and the shared Session Context is the Sink. This mechanism supports synchronous multi-party interaction.

The interaction control system can be used to set up different multi-user interaction scenarios. For example, in the Free Interaction mode, only users with Modify capability can alter Session State by creating or modifying drawings. Users with only Browse capability can independently apply viewing transformations like panning and zooming. Users with only Access capability simply observe shared interaction on the canvas. In the Regulated Interaction mode, users take turns to use the canvas to edit primitive objects and change Session State. Access regulation methods can be used for fine grained interaction control. Permissions attached to drawings and primitive objects regulate what operations users can perform on those objects. For example, a user can protect something he draws by removing the Modify permission of that object.

Sha-Draw is built on the Structural and Media Models. and uses the Collaborative Graphics Substrate of the Shastra environment. This high level of abstraction, and the common messaging system, lets it interoperate with any tool in the environment that speaks the same command language. The Collaborative Graphics Substrate provides a 2D Graphics Widget Stub, which encapsulates the command language and communication. Any tool can instantiate a stub. It can then participate in the collaborative process in a media-aware manner by using its own mechanisms for local drawing and interaction, or by using substrate facilities to do the same. The substrate also provides a 2D Graphics Widget that implements canvases and is bundled with interaction functionality. Tools can instantiate a widget for media-unaware interoperation.

A.2 Sha-Poly

Sha-Poly is an Agent in the Shastra environment, built on the Structural and Media Models. (The notion of Agents, and these Models, is described in Section 2.) It uses structured 3D graphics, models, and the associated interaction as its media type.

Sha-Poly is used as a visualization and graphical-object browsing and manipulation tool. It allows graphical objects to be manipulated and visualized in platform independent XS graphics windows. XS based tools transparently use available hardware graphics facilities whenever available.

Sha-Poly allows the user to create and use multiple graphics canvases (Contexts in the Model). Models can be moved to and from different canvases using interface facilities. It supports input and output of 3D models from files, and understands a variety of 3D model representation formats. Models are the data objects that it manipulates. Figure A.3 depicts the block architecture of Sha-Poly.

In Sha-Poly, the actual process of 3D graphical interaction is captured using an embedded command language. This interaction, containing data with temporal attributes, constitutes a media stream. Sources for this media stream can be the local

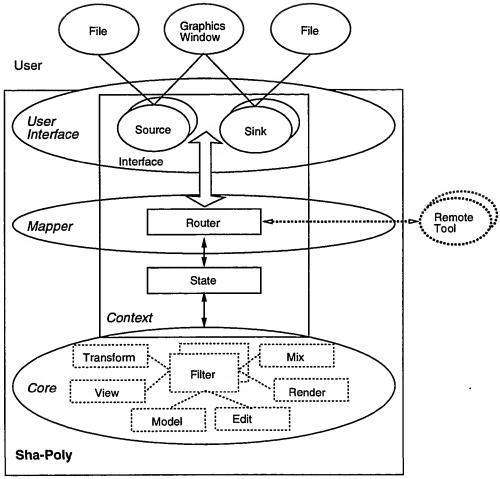

Figure A.3 High Level Architecture of Sha-Poly

graphics canvas, local files, or remote tools. Sinks for this media stream can be the local canvas, local files, or remote tools. The embedded command language can be used as a scripting language for simple animations using 3D graphics. Filters allow the application of 3D transformations, illumination and lighting model specification, material property specification and stream mixing.

Graphical interaction is recorded in a file by setting up the canvas as a Source, and the file as a Sink. Interaction is played back from a file by setting up the file as a Source, and the canvas as a Sink.

Sha-Poly supports graphical interaction in a distributed setting, by interoperating with other tools on heterogeneous platforms. Support for heterogeneous graphics is enabled by XS (see Section 3.2.3). Sha-Poly allows a user to interact in canvases of other Sha-Poly instances running on different machines across the network. This is done by setting up the local canvas as a Source, and the remote canvases as Sinks. Alternately, the user plays back recorded interaction files into multiple remote canvases by setting up the local file as a Source, and the local and remote canvases as Sinks. Only the user controlling the Source can draw into the canvas, though everyone sees the graphical interaction.

In the collaborative setting, Sha-Poly is used as a multi-user 3D graphics system. A collaborative session consisting of Sha-Poly instances lets a group of collaborators synchronously interact over a shared 3D canvas. It supports shared viewing of 3D models using different display and visualization techniques. When a user joins the session, Sha-Poly creates a shared canvas. 3D models are introduced into the shared canvas using interface facilities. These data objects are transported to all participating sites. Graphical interaction with the models is shared, by redirecting the input media stream to the shared Session Context, i.e. the new canvas is the Source and the shared Session Context the Sink. This mechanism supports synchronous multi-party interaction.

The typical user interface for collaboration is shown in Figure A.4. It depicts the shared canvas (at bottom) with a shared object. Also shown are telepointers,

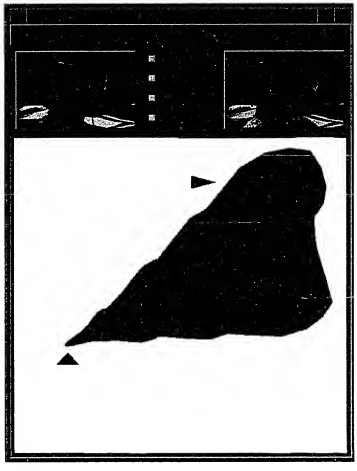

Figure A.4 Sha-Poly User Interface

which are typically used to point at features of the shared model, and for gesturing. The upper part of the figure shows an example of integrating a Video Widget, which allows for transparent, media-unaware interoperation with a Video Agent. It depicts the participating users.

The interaction control system can be used to set up different kinds of multi-user interaction scenarios. For example, in the Free Interaction mode, only users with Modify capability can alter Session State by changing illumination parameters, or viewing and modeling parameters of the scene. Users with only Browse capability can independently apply viewing transformations and modeling transformations, allowing them to maintain independent views of the shared state. Users with only Access capability simply observe shared interaction on the canvas. In the Regulated Interaction mode, users take turns to use the canvas to change Session State.

Access regulation methods can be used for fine grained interaction control. Permissions attached to models regulate what operations users can perform on them. For example, a user can disallow others from changing material properties of a model, or its 3D location, removing its Modify.

Sha-Poly is built on the Structural and Media Models. and uses the Collaborative Graphics Substrate of the Shastra environment. This high level of abstraction lets it interoperate with any tool in the environment that speaks the same command language. The Collaborative Graphics Substrate provides a 3D Graphics Widget Stub, which encapsulates the command language, and communication. Any tool can instantiate a stub. It can then participate in the collaborative process in a media-aware manner by using its own mechanisms for display and interaction, or by using substrate facilities to do the same. The substrate also provides a 3D Graphics Widget that implements canvases and is bundled with interaction and display functionality on top of XS. Tools can instantiate a widget for media-unaware interoperation.

Appendix B: Multimedia Support

B.1 Talk

Sha-Talk is a very simple Agent in the Shastra environment, built on the Structural and Media Models. (The notion of Agents, and of these Models, is described in Section 2.) Sha-Talk uses text as its media type.

Sha-Talk is used as a text notepad. Sha-Talk allows the user to create and use multiple canvases (Contexts in the Model). Text can be moved to and from different canvases using interface facilities. Sha-Talk supports input and output of text from files. They are the data objects that it manipulates. Figure B.1 depicts the block architecture of Sha-Talk.

In Sha-Talk, textual interaction is captured using an embedded command language. This interaction is basically textual data with some control information, and constitutes a media stream. Sources for this media stream can be the local canvas, local files, or remote sources like other tools. Sinks for this media stream can be the local canvas, local files, or remote sinks like other tools.

Text interaction is recorded into files by setting up the canvas as a Source, and the file as a Sink. It is played back from files by setting up the file as a Source, and the canvas as a Sink. The interface provides transport control facilities.

In the distributed setting, Sha-Talk allows a user to write into canvases of other Sha-Talk instances running on different machines across the network. This is done by setting up the local canvas as a Source, and the remote canvases as Sinks. Alternately, the user plays back recorded files into multiple remote canvases by setting up the local file as a Source, and the local and remote canvases as Sinks. Only the user controlling the Source can type into the canvas, though everyone sees the interaction. A telepointing facility is implemented, and exploited for gesturing and pointing.

In the collaborative setting, Sha-Talk is used as a multi-user notepad. A collaborative session consisting of Sha-Talk instances lets a group of collaborators synchronously communicate using text. When a user joins the session, Sha-Talk creates

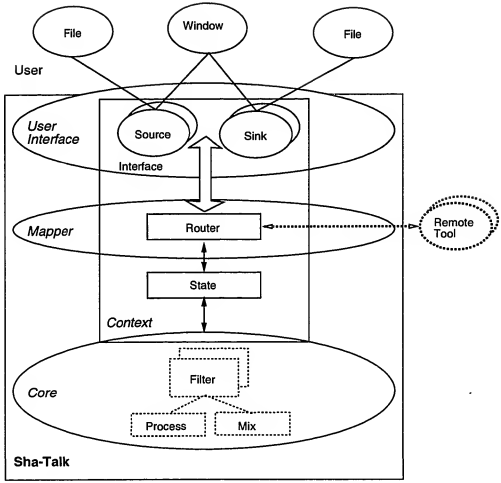

Figure B.1 High Level Architecture of Sha-Talk

a a panel in a shared canvas. Interaction streams from all sites are rendered into their respective panel on this canvas. The interaction is shared, by redirecting the input media stream to the shared Session Context, i.e. the new canvas is the Source and the shared Session Context is the Sink. This mechanism supports synchronous multi-party interaction. It is particularly useful when we do not have multimedia communication facilities on the desktop.

The interaction control system can be used to set up different multi-user interaction scenarios. For example, in the Free Interaction mode, only users with Modify capability can alter Session State by sending it text. Users with only Browse capability can independently apply viewing transformations like scrolling. Users with only Access capability simply observe shared interaction on the canvas.

The typical user interface is shown in Figure B.2. It depicts a canvas with two text panels. Bitmap images of users are used to identify owners of panels.

Sha-Talk is built on the Structural and Media Models. and uses the Collaboration Substrate of the Shastra environment. This high level of abstraction, and the common messaging system, lets it interoperate with any tool in the environmant that speaks the same command language. The Collaboration Substrate provides a Text Widget Stub, which encapsulates the command language, and communication. Any tool can instantiate a stub. It can then participate in the collaborative process in a media-aware manner by using its own mechanisms for local interaction, or by using substrate facilities to do the same. The substrate also provides a Text Widget that implements canvases and is bundled with interaction functionality. Tools can instantiate a widget for media-unaware interoperation.

Sha-Talk provides the infrastructure to implement multi-user text editors and text processing systems.

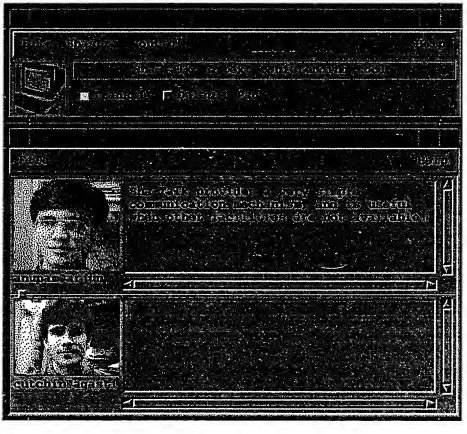

Figure B.2 Sha-Talk User Interface

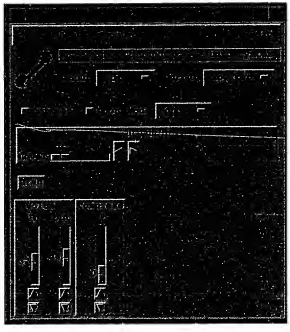

Figure B.3 Sha-Phone User Interface

B.2 Sha-Phone

Sha-Phone is an Audio Agent in the Shastra environment, built on the Structural and Media Models. (The notion of Agents, and these Models, is described in Section 2.) Sha-Phone uses digital audio as its media type.

Sha-Phone is used as an audio processing tool, and facilitates capture, playback and processing of audio signals. It provides different kinds of filtering, and special effects, and supports audio transformations like mixing, pitch-shifting and amplitude adjustment. The Motif-based [67] user interface is shown in Figure B.3.

Sha-Phone allows the user to create and use multiple audio contexts, which are logical representations of external devices like microphones, speakers etc. Audio objects can be moved to and from different contexts using interface facilities. Sha-Phone supports input and output of audio clips from files. They are the data objects that it manipulates. Figure B.4 depicts the block architecture of Sha-Phone.

In Sha-Phone, audio interaction is captured using an embedded command language. This interaction is basically audio data with some control information, and constitutes a media stream. Sources for this media stream can be the local audio context, local files, or remote sources like other tools. Sinks for this media stream can be the local audio context, local files, or remote sinks like other tools. The embedded command language can be used as a scripting language for dynamic control of audio rendition. Filters allow the implementation of special effects, pitch shifting and amplitude adjustment, as well as stream mixing.

Audio is recorded into files by setting up the microphone as a Source, and the file as a Sink. They are played back from files by setting up the file as a Source, and local speakers as a Sink. The interface provides transport control and filtering facilities.

In the distributed setting, Sha-Phone allows a user to redirect audio signals into audio contexts of other Sha-Phone instances running on different machines across the network. This is done by setting up the local context as a Source, and the remote contexts as Sinks. Alternately, the user plays back recorded files into multiple remote contexts by setting up the local file as a Source, and the local and remote canvases as Sinks. Only the user controlling the Source can control rendition *via* transport control or filters. Everyone else just receives the audio stream.

In the collaborative setting, Sha-Phone is used as a multi-user audio processing system. A collaborative session consisting of Sha-Phone instances lets a group of collaborators conduct an audio conference. When a user joins the session, Sha-Phone creates a shared context. Audio interaction streams from all sites are mixed and rendered via this context. Thus Sha-Phone can be used as a desktop audio conferencing tool. Audio objects are shared by transporting them to all sites. Audio interaction is shared, by redirecting the input media stream to the shared Session Context, i.e. the new context is the Source and the shared Session Context is the Sink. Sha-Phone can operate as a collaborative audio manipulation system. This mechanism supports synchronous multi-party manipulation of audio objects. Besides supporting multi-point

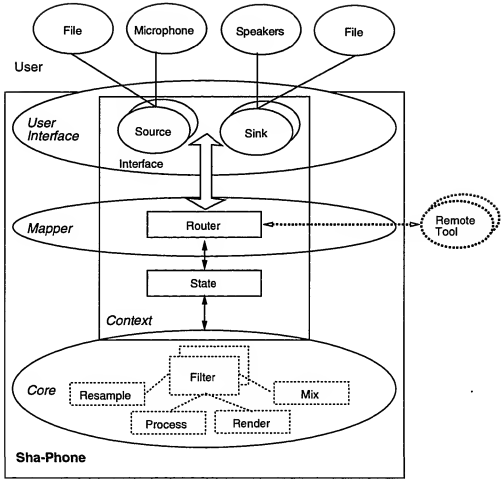

Figure B.4 High Level Architecture of Sha-Phone

recording and playback, the conferenced system allows collaborative manipulation of live and stored audio streams – interaction with reviewability.

The interaction control system can be used to set up different multi-user interaction scenarios. For example, in the Free Interaction mode, only users with Modify capability can alter Session State by sending it audio data or streams. Users with only Access capability simply receive shared audio in the context. In the Regulated Interaction mode, users take turns to use the context to broadcast audio.

Sha-Phone is built on the Structural and Media Models. and uses the Collaborative Audio Substrate of the Shastra environment. This high level of abstraction, and the common messaging system, lets it interoperate with any tool in the environment that speaks the same command language. The Collaborative Graphics Substrate provides an Audio Widget Stub, which encapsulates the command language, and communication. Any tool can instantiate a stub. It can then participate in the collaborative process in a media-aware manner by using its own mechanisms for audio presentation and control, or by using substrate facilities to do the same. The substrate also provides a Audio Widget that implements contexts and is bundled with interaction functionality. Tools can instantiate a widget for media-unaware interoperation. Thus inter-operable with other Shastra tools, Sha-Phone is used to record and playback audio information stored in multimedia objects maintained by other tools, by setting up appropriate contexts and filters.

B.3 Video

Sha-Video is a Video Agent in the Shastra environment, built on the Structural and Media Models. (The notion of Agents, and these Models, is described in Section 2.) Sha-Video uses still images and live and stored video streams as its media type.

Sha-Video is a video processing toolkit that supports video recording and playback (without sound), as well as image processing. The typical user interface is shown in Figure B.5. It depicts a control panel (at left) that is used to create two video canvases

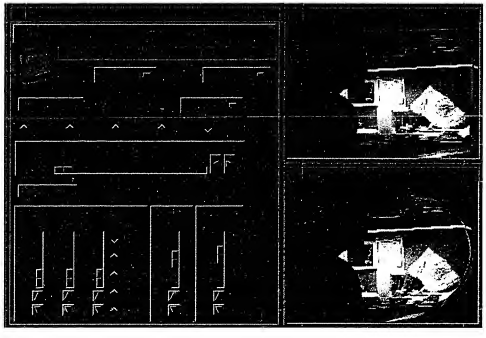

Figure B.5 Sha-Video User Interface

(at right) that show unfiltered (at top-right) and filtered (at bottom-right) playback of a stored video stream.

Sha-Video allows the user to create and use multiple video contexts. Video contexts are abstractions that encapsulate the notion of external devices like cameras, video cassette players and recorders, frame grabbers for television signal capture, or simply canvases for desktop visual output. Sha-Video supports input and output of images and video clips from files. Images and video clips, can be moved to and from different contexts using interface facilities. They are the data objects that it manipulates. Figure B.6 depicts the block architecture of Sha-Video.

In Sha-Video, the actual process of video interaction is captured using an embedded command language. This interaction contains video and control data with temporal attributes and constitutes a media stream. Sources for this media stream can be the video context, local files, or remote sources like other tools. Sinks for this media stream can be the video context, local files, or remote sinks like other tools. The embedded command language can be used as a scripting language for simple video stream control. Filters allow the application of 2D transformations, color changes, image processing, and special effects.

Video is recorded into files by setting up a video context as a Source, and the file as a Sink. It is played back from files by setting up the file as a Source, and the video context as a Sink. The user interface provides transport control facilities, as well as filtering and image processing control.

In the distributed setting, Sha-Video allows a user to display video into video contexts of other Sha-Video instances running on different machines across the network. This is done by setting up the local video context as a Source, and the remote video contexts as Sinks. Alternately, the user plays back recorded files into multiple remote video contexts by setting up the local file as a Source, and the local and remote video contexts as Sinks. Only the user controlling the Source can control the video stream, though everyone sees the images and interaction. A telepointing facility is provided, and is exploited for gesturing and pointing to features in the video images.

In the collaborative setting, Sha-Video is used as a multi-user video and image processing system. Sha-Video can be used as a desktop video conferencing tool A collaborative session consisting of Sha-Video instances lets a group of collaborators synchronously conduct a silent video conference. Sha-Video also serves as a collaborative video and image manipulation system. allowing multiple users to edit images on shared virtual whiteboards. When a user joins the session, Sha-Video creates a shared video context. Video and interaction streams from all sites are rendered into this video context. Video objects that are introduced into this context are shared with every other user in the session, by transporting the underlying data object. Video streams are shared by redirecting the input media stream to the shared Session Context, i.e. the new context is the Source and the shared Session Context is the Sink. This mechanism supports synchronous multi-party interaction. Besides supporting multi-point recording and playback, the conferenced system allows collaborative manipulation of live and stored video streams and still images, providing interaction with reviewability.

The interaction control system can be used to set up different multi-user interaction scenarios. For example, in the Free Interaction mode, only users with Modify capability can alter Session State by sending it video streams or objects. Users with only Access capability simply observe shared interaction in the video context. In the Regulated Interaction mode, users take turns to edit objects and change Session State. Access regulation methods can be used for fine grained interaction control. Permissions attached to drawings and primitive objects regulate what operations users can perform on those objects. For example, a user can protect a shared image by removing the Modify permission of that object.

Sha-Video is built on the Structural and Media Models. and uses the Collaborative Video Substrate of the Shastra environment. This high level of abstraction, and the common messaging system, lets it interoperate with any tool in the environment that speaks the same command language. The Collaborative Graphics Substrate provides a Video Widget Stub, which encapsulates the command language,

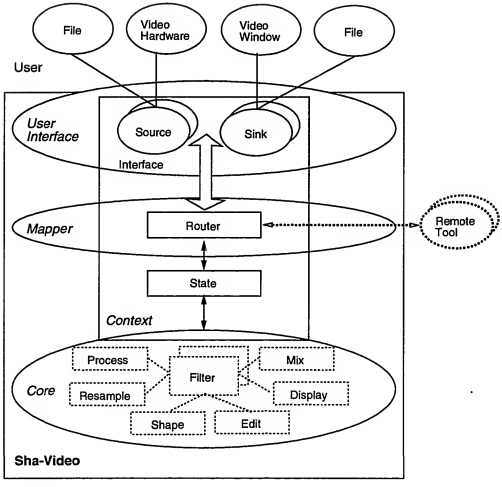

Figure B.6 High Level Architecture of Sha-Video

and communication. Any tool can instantiate a stub. It can then participate in the collaborative process in a media-aware manner by using its own mechanisms for local drawing and interaction, or by using substrate facilities to do the same. The substrate also provides a Video Widget that implements video contexts and is bundled with interaction functionality. Tools can instantiate a widget for media-unaware interoperation. Sha-Video inter-operates in the Shastra environment and is used by other tools, to playback and record video information stored in multimedia objects that they manipulate.

Appendix C: Geometric Modeling Support

C.1 Shilp

Shilp is a Geometric Modeling Agent in the Shastra environment, built on the Structural and Media Models. (The notion of Agents, and these Models, is described in Section 2.) Shilp uses solid models and modeling interaction as its media type.

Shilp is a boundary representation based geometric modeling system. Current functionality of the toolkit includes extrude, revolve and offset operations, edit operations on solids and laminas, pattern matching and replacement, boolean set operations and assembly, fleshing of wireframes with smooth algebraic surface patches, and blending and rounding of solid corners and edges. Shilp provides mechanisms for creating complex solid models from simple ones. The typical user interface is shown in Figure C.2. It depicts a control panel (at left) and an XS based modeling context, with some geometric models. The components of Shilp are depicted in Figure C.1.

Shilp allows the user to create and use multiple modeling contexts (Contexts in the Model). Models can be moved to and from different contexts using interface facilities. Shilp supports input and output of geometric models from files, and understands a variety of 3D model representation formats. Models are the data objects that it manipulates. Figure C.3 depicts the block architecture of Shilp.

In Shilp, the actual interactive process of creation of models is captured using an embedded command language. This interaction contains data with temporal attributes and constitutes a media stream. Sources for this media stream can be the actual modeling context, local files, or remote sources like other tools. Sinks for this media stream can be the local context, local files, or remote sinks like other tools. Filters allow the application of transformations and editing operations, and stream mixing.

Modeling interaction is recorded into files by setting up the context as a Source, and the file as a Sink. Interaction is played back from files by setting up the file as

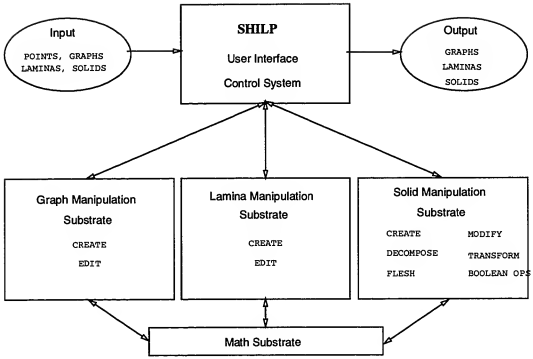

Figure C.1 Components of Shilp

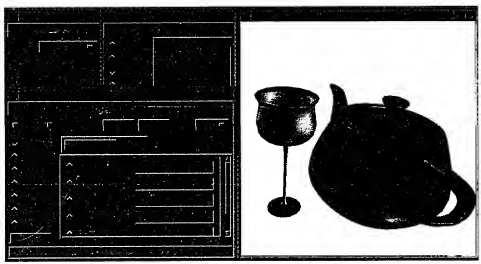

Figure C.2 Shilp User Interface

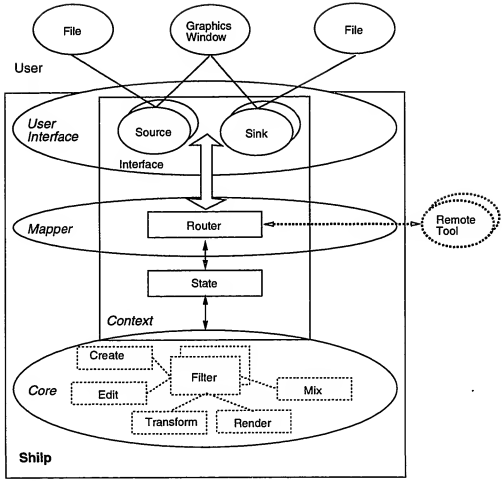

Figure C.3 Application Architecture of Shilp

a Source, and the canvas as a Sink. The interface provides basic transport control facilities for the media stream.

Shilp supports graphical interaction in a distributed setting, by interoperating with other tools on heterogeneous platforms. Support for heterogeneous graphics is enabled by XS (see Section 3.2.3). In the distributed setting, Shilp allows a user to redirect modeling interaction into contexts of other Shilp instances running on different machines across the network. This is done by setting up the local context as a Source, and the remote contexts as Sinks. Alternately, the user plays back recorded interaction files into multiple remote contexts by setting up the local file as a Source, and the local and remote contexts as Sinks. Only the user controlling the Source can interact in the context, though everyone sees the interaction and modeling. A telepointing facility is implemented, and is exploited for gesturing and pointing.

In the collaborative setting, Shilp is used as a multi-user geometric modeling system. A collaborative session consisting of Shilp instances lets a group of collaborators synchronously create and edit geometric designs. When a user joins the session, Shilp creates a shared context. Modeling and interaction streams from all sites are mixed and rendered into this context. In the simple implementation, only models that are introduced into this canvas by modeling operations are shared with every other user in the session, by transporting the underlying data object. In the more complex case, the actual interaction involved in modeling is shared, by redirecting the input media stream to the shared Session Context, i.e. the new context is the Source and the shared Session Context is the Sink. This mechanism supports synchronous multi-party interaction.

The interaction control system can be used to set up different multi-user interaction scenarios. For example, in the Free Interaction mode, only users with Modify capability can alter Session State by creating or modifying models. Users with only Browse capability can independently apply viewing transformations to the 3D scene. Users with only Access capability simply observe shared interaction in the context. In the Regulated Interaction mode, users take turns to use the context to create and

edit models and change Session State. Access regulation methods can be used for fine grained interaction control. Permissions attached to models regulate what operations users can perform on those objects. For example, a user can protect a model by removing the Modify permission of that object.

Shilp is built on the Structural and Media Models. and uses the Collaborative Graphics Substrate of the Shastra environment. This high level of abstraction, and the common messaging system, lets it interoperate with any tool in the environment that speaks the same command language.

Shilp is described in detail in [7].

Appendix D: Collaborative Games

D.1 Sha-Chess

Sha-Chess supports a shared virtual 3D chess board and typifies virtual collaborative environments for games and entertainment-oriented interaction. It demonstrates how any structured data with well defined interaction semantics can be treated as a media type in the Media Model of Shastra.

Sha-Chess is an Agent in the Shastra environment, built on the Structural and Media Models. (The notion of Agents, and these Models, is described in Section 2.) Sha-Chess uses interaction over a virtual chess board as its media type.

As a stand-alone application, it provides a 3D graphical interface on which chess games can be played. It is built on top of XS, a hardware independent 3D graphics system (see Section 3.2.3). Sha-Chess lets a user play against a chess playing program, or against another user, locally. It supports a regulated mode where it allows only legal moves, and enforces turns to make moves. It also supports an unregulated mode, where the system just provides a game playing surface without regulating interaction, much like a physical chess board. The typical user interface is shown in Figure D.1. It depicts a control panel (at top-left) with bitmap images of the players, and a graphics window (at right) that shows the status of the chess game in progress. The other graphics window (at bottom-left) shows an alternate view of the same board (the other player's view in this case).

Sha-Chess allows the user to create and use multiple contexts, which are essentially virtual chess boards. Games can be moved to and from different contexts using interface facilities. Sha-Chess supports input and output of chess games from files. They are the data objects that it manipulates. Figure D.2 depicts the block architecture of Sha-Chess.

In Sha-Chess, the actual interactive process of playing a game is captured using an embedded command language. This interaction contains data with temporal attributes and constitutes a media stream. Sources for this media stream can be the

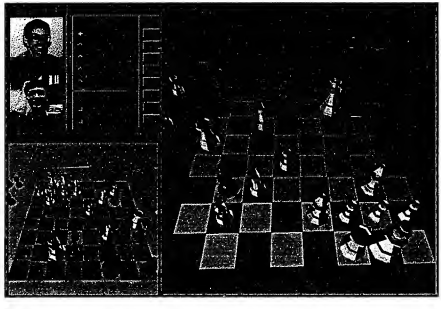

Figure D.1 Sha-Chess User Interface

actual canvas, local files, or remote sources like other tools, and chess playing programs. Sinks for this media stream can be the local canvas, local files, or remote sinks like other tools and chess playing programs.

Chess games are recorded into files by setting up the context as a Source, and the file as a Sink. They are played back from files by setting up the file as a Source, and the context as a Sink. The interface provides transport control facilities for playing back games.

In the distributed setting, Sha-Chess allows a user to show other users a chess game by redirecting the live local stream into contexts of other Sha-Chess instances running on different machines across the network. This is done by setting up the local context as a Source, and the remote contexts as Sinks. Alternately, the user plays back recorded games into multiple remote contexts by setting up the local file as a Source, and the local and remote contexts as Sinks. Only the user controlling the Source can interact with the context, though everyone sees the interaction as the game progresses.

In the collaborative setting, Sha-Chess is used as a shared multi-user chess board. A collaborative session consisting of Sha-Chess instances lets a group of collaborators play chess games on a shared chess board. When a user joins the session, Sha-Chess creates a shared context. Interaction streams from all sites are mixed and rendered into this context. In the simple implementation, only actual moves are transmitted to every other user in the session, by transporting the appropriate control data. In the more complex case, the all interaction involved in making the move, *e.g.* picking a piece and placing it, is shared by redirecting the input media stream to the shared Session Context, i.e. the new context is the Source and the shared Session Context is the Sink. This mechanism supports synchronous multi-party interaction. A collaboration of Sha-Chess instances thus creates a virtual world and provides an interface that lets a group of geographically separated chess players synchronously interact over a shared virtual chess board.

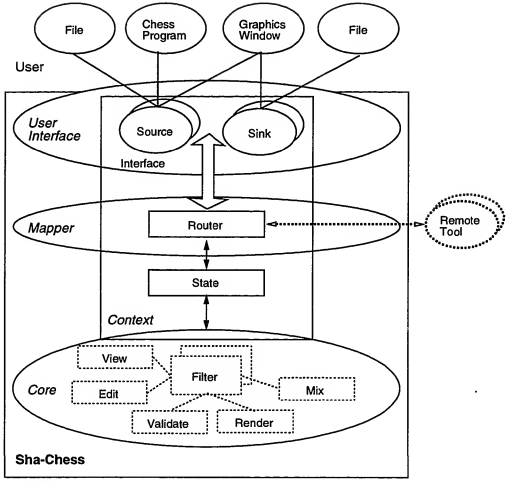

Figure D.2 High Level Architecture of Sha-Chess

Sha-Chess exploits the interaction control mechanism of Shastra to support a variety of modes in which the multiple users interact in the virtual environment.

At one extreme, Sha-Chess performs no move or turn regulation. It simply transmits moves made by different players who have Modify permission and updates the view at all sites with Access permission. Using audio, video and text communication channels to coordinate matters, users can play a game successfully in this mode. Alternately, if exactly two people are given Modify permission for the session, and they would be the only active participants, with everyone else getting a current view of the board. If Sha-Chess is also switched to regulated mode, allowing only legal moves in turn, a tournament situation is simulated in this virtual environment. Alternately, the group of users can be divided into two teams such that any member of a team can make a move for that team. In yet another scenario, using the Regulated Interaction mode for the collaborative session, a single user can to teach others fundamentals of the game of chess, or discuss strategy.

Sha-Chess is built on the Structural and Media Models. and uses the Collaborative Graphics Substrate of the Shastra environment.

VITA

VITA

Vinod Anupam was born on the twenty-first of October 1967, in Meerut, India. He cleared the All India Secondary School Examination (10^{th} Grade) from Sainik School, Ghorakhal (Nainital) in March 1982 and the All India Senior School Certificate Examination (12^{th} Grade) from Kendriya Vidyalaya, Jalahalli (Bangalore) in March 1984. He obtained a bachelor's degree in Computer Science from Birla Institute of Technology and Science (BITS), Pilani in May 1988. He joined the Ph.D. program in the Department of Computer Sciences of Purdue University in August 1988 and was awarded the Ph.D. in August 1994. He is a member of Upsilon Pi Epsilon.

His research interests include computer-supported cooperative work and groupware, networking and distributed systems, geometric modeling, computer aided design and concurrent engineering, graphics and visualization, multimedia, and graphical user interfaces.

```
/*****************************************************************************
    ***/
/*****************************************************************************
    ***/
/**
    **/
/** This SHASTRA software is not in the Public Domain. It is distributed on
    **/
/** a person to person basis, solely for educational use and permission is
    **/
/** NOT granted for its transfer to anyone or for its use in any commercial
    **/
/** product.  There is NO warranty on the available software and neither
    **/
/** Purdue University nor the Applied Algebra and Geometry group directed
    **/
/** by C.  Bajaj accept responsibility for the consequences of its use.
    **/
/**
    **/
/*****************************************************************************
    ***/
/*****************************************************************************
    ***/
#include <stdio.h>

#include <shastra/datacomm/audioBiteH.h>
#include <shastra/network/mplex.h>
#include <shastra/network/rpc.h>

#define STANDALONEnn

int
audioBiteOut(fd, pABite)
    int             fd;
    audioBite      *pABite;
{
    XDR             xdrs;
    int             retVal = 0;

#ifdef STANDALONE
    {
        FILE           *fp;
        fp = stdout /* fdopen(fd,"w") */ ;
        xdrstdio_create(&xdrs, fp, XDR_ENCODE);
        if (!xdr_audioBite(&xdrs, pABite)) {
            retVal = -1;
        }
    }
#else               /* STANDALONE */
    /*
     * xdrstdio_create(mplexXDRSEnc(fd), mplexOutStream(fd), XDR_ENCODE);
     */
```

```
    if (!xdr_audioBite(mplexXDRSEnc(fd), pABite)) {
        retVal = -1;
    }
#endif              /* STANDALONE */
    return retVal;
}

int
audioBiteIn(fd, pABite)
    int             fd;
    audioBite      *pABite;
{
    XDR             xdrs;
    int             retVal = 0;

    audioBiteXDRFree(pABite);
#ifdef STANDALONE
    {
        FILE           *fp;
        fp = stdin /* fdopen(fd,"r") */ ;
        xdrstdio_create(&xdrs, fp, XDR_DECODE);
        if (!xdr_audioBite(&xdrs, pABite)) {
            retVal = -1;
        }
    }
#else               /* STANDALONE */
    /*
     * xdrstdio_create(mplexXDRSDec(fd), mplexInStream(fd), XDR_DECODE);
     */
    if (!xdr_audioBite(mplexXDRSDec(fd), pABite)) {
        retVal = -1;
    }
#endif              /* STANDALONE */
    return retVal;
}

int
audioBiteMemOut(buf, size, pABite)
    char           *buf;
    int             size;
    audioBite      *pABite;
{
    XDR             xdrs;
    int             retVal = 0;

    xdrmem_create(&xdrs, buf, size, XDR_ENCODE);
    if (!xdr_audioBite(&xdrs, pABite)) {
        retVal = -1;
    }
    xdr_destroy(&xdrs);
    return retVal;
}
```

```
int
audioBiteMemIn(buf, size, pABite)
    char           *buf;
    int             size;
    audioBite      *pABite;
{
    XDR             xdrs;
    int             retVal = 0;

    audioBiteXDRFree(pABite);
    xdrmem_create(&xdrs, buf, size, XDR_DECODE);
    if (!xdr_audioBite(&xdrs, pABite)) {
        retVal = -1;
    }
    xdr_destroy(&xdrs);
    return retVal;
}

int
audioBitesOut(fd, pABites)
    int             fd;
    audioBites     *pABites;
{
    XDR             xdrs;
    int             retVal = 0;

#ifdef STANDALONE
    {
        FILE           *fp;
        fp = stdout /* fdopen(fd,"w") */ ;
        xdrstdio_create(&xdrs, fp, XDR_ENCODE);
        if (!xdr_audioBites(&xdrs, pABites)) {
            retVal = -1;
        }
    }
#else               /* STANDALONE */
    /*
     * xdrstdio_create(mplexXDRSEnc(fd), mplexOutStream(fd), XDR_ENCODE);
     */
    if (!xdr_audioBites(mplexXDRSEnc(fd), pABites)) {
        retVal = -1;
    }
#endif              /* STANDALONE */
    return retVal;
}

int
audioBitesIn(fd, pABites)
    int             fd;
    audioBites     *pABites;
{
    XDR             xdrs;
    int             retVal = 0;
```

```
    audioBitesXDRFree(pABites);
#ifdef STANDALONE
    {
        FILE           *fp;
        fp = stdin /* fdopen(fd,"r") */ ;
        xdrstdio_create(&xdrs, fp, XDR_DECODE);
        if (!xdr_audioBites(&xdrs, pABites)) {
            retVal = -1;
        }
    }
#else               /* STANDALONE */
    /*
     * xdrstdio_create(mplexXDRSDec(fd), mplexInStream(fd), XDR_DECODE);
     */
    if (!xdr_audioBites(mplexXDRSDec(fd), pABites)) {
        retVal = -1;
    }
#endif              /* STANDALONE */
    return retVal;
}

int
audioBitesMemOut(buf, size, pABites)
    char           *buf;
    int             size;
    audioBites     *pABites;
{
    XDR             xdrs;
    int             retVal = 0;

    xdrmem_create(&xdrs, buf, size, XDR_ENCODE);
    if (!xdr_audioBites(&xdrs, pABites)) {
        retVal = -1;
    }
    xdr_destroy(&xdrs);
    return retVal;
}

int
audioBitesMemIn(buf, size, pABites)
    char           *buf;
    int             size;
    audioBites     *pABites;
{
    XDR             xdrs;
    int             retVal = 0;

    audioBitesXDRFree(pABites);
    xdrmem_create(&xdrs, buf, size, XDR_DECODE);
    if (!xdr_audioBites(&xdrs, pABites)) {
        retVal = -1;
    }
```

```
    xdr_destroy(&xdrs);
    return retVal;
}

int
audioClipOut(fd, pAClip)
    int             fd;
    audioClip     *pAClip;
{
    XDR             xdrs;
    int             retVal = 0;

#ifdef STANDALONE
    {
        FILE           *fp;
        fp = stdout /* fdopen(fd,"w") */ ;
        xdrstdio_create(&xdrs, fp, XDR_ENCODE);
        if (!xdr_audioClip(&xdrs, pAClip)) {
            retVal = -1;
        }
    }
#else               /* STANDALONE */
    /*
     * xdrstdio_create(mplexXDRSEnc(fd), mplexOutStream(fd), XDR_ENCODE);
     */
    if (!xdr_audioClip(mplexXDRSEnc(fd), pAClip)) {
        retVal = -1;
    }
#endif              /* STANDALONE */
    return retVal;
}

int
audioClipIn(fd, pAClip)
    int             fd;
    audioClip     *pAClip;
{
    XDR             xdrs;
    int             retVal = 0;

    audioClipXDRFree(pAClip);
#ifdef STANDALONE
    {
        FILE           *fp;
        fp = stdin /* fdopen(fd,"r") */ ;
        xdrstdio_create(&xdrs, fp, XDR_DECODE);
        if (!xdr_audioClip(&xdrs, pAClip)) {
            retVal = -1;
        }
    }
#else               /* STANDALONE */
    /*
     * xdrstdio_create(mplexXDRSDec(fd), mplexInStream(fd), XDR_DECODE);
```

```
     */
    if (!xdr_audioClip(mplexXDRSDec(fd), pAClip)) {
        retVal = -1;
    }
#endif              /* STANDALONE */
    return retVal;
}

int
audioClipMemOut(buf, size, pAClip)
    char           *buf;
    int             size;
    audioClip     *pAClip;
{
    XDR             xdrs;
    int             retVal = 0;

    xdrmem_create(&xdrs, buf, size, XDR_ENCODE);
    if (!xdr_audioClip(&xdrs, pAClip)) {
        retVal = -1;
    }
    xdr_destroy(&xdrs);
    return retVal;
}

int
audioClipMemIn(buf, size, pAClip)
    char           *buf;
    int             size;
    audioClip     *pAClip;
{
    XDR             xdrs;
    int             retVal = 0;

    audioClipXDRFree(pAClip);
    xdrmem_create(&xdrs, buf, size, XDR_DECODE);
    if (!xdr_audioClip(&xdrs, pAClip)) {
        retVal = -1;
    }
    xdr_destroy(&xdrs);
    return retVal;
}

void
freeAudioBite(pABite)
    audioBite      *pABite;
{
    if (pABite == NULL) {
        return;
    }
    if (pABite->data.data_val != NULL) {
        free(pABite->data.data_val);
    }
```

```
    memset(pABite, 0, sizeof(audioBite));
}


void
freeAudioBites(pABites)
    audioBites     *pABites;
{
    audioBitesXDRFree(pABites);
}

audioBite      *
copyAudioBite(pABite, destpABite)
    audioBite      *pABite;
    audioBite      *destpABite;
{
    audioBite      *newpABite;
    int             i;

    if (pABite == NULL) {
        return NULL;
    }
    if (destpABite == NULL) {
        newpABite = (audioBite *) malloc(sizeof(audioBite));
    } else {
        newpABite = destpABite;
    }

    memcpy(newpABite, pABite, sizeof(audioBite));
    newpABite->data.data_val = (char *) malloc(newpABite->data.data_len *
                        sizeof(newpABite->data.data_val[0]));
    memcpy(newpABite->data.data_val,pABite->data.data_val,
        newpABite->data.data_len * sizeof(newpABite->data.data_val[0]));
    return newpABite;
}

audioBites     *
copyAudioBites(pABites, destpABites)
    audioBites     *pABites;
    audioBites     *destpABites;
{
    int             i;
    audioBites     *newpABites;

    if (pABites == NULL) {
        return NULL;
    }
    if (destpABites == NULL) {
        newpABites = (audioBites *) malloc(sizeof(audioBites));
    } else {
        newpABites = destpABites;
    }
    memcpy(newpABites, pABites, sizeof(audioBites));
```

```
    newpABites->audioBites_val = (audioBite *)
        malloc(newpABites->audioBites_len * sizeof(audioBite));
    for (i = 0; i < newpABites->audioBites_len; i++) {
        copyAudioBite(&pABites->audioBites_val[i],
            &newpABites->audioBites_val[i]);
    }
    return newpABites;
}


void
inputAudioBite(fp, pABite)
    FILE           *fp;
    audioBite      *pABite;
{
    int             i, n;

    fscanf(fp, "%ld", &pABite->lIdTag);
    fscanf(fp, "%ld", &pABite->lSIdTag);
    fscanf(fp, "%ld", &pABite->lPerms);
    fscanf(fp, "%hd", &pABite->biteFormat);
    fscanf(fp, "%hd", &pABite->biteComp);
    fscanf(fp, "%ld", &pABite->biteMode);
    fscanf(fp, "%ld", &pABite->biteSize);
    fscanf(fp, "%ld", &pABite->biteRate);
    fscanf(fp, "%ld", &pABite->data.data_len);
    pABite->data.data_val = (char *) malloc(pABite->data.data_len *
                        sizeof(pABite->data.data_val[0]));
    for (i = 0; i < pABite->data.data_len; i++) {
        fscanf(fp, "%hd", &n);
        pABite->data.data_val[i] = n;
    }
}

void
outputAudioBite(fp, pABite)
    FILE           *fp;
    audioBite      *pABite;
{
    int             i;

    fprintf(fp, "%ld\n", pABite->lIdTag);
    fprintf(fp, "%ld\n", pABite->lSIdTag);
    fprintf(fp, "%ld\n", pABite->lPerms);
    fprintf(fp, "%hd\n", pABite->biteFormat);
    fprintf(fp, "%hd\n", pABite->biteComp);
    fprintf(fp, "%ld\n", pABite->biteMode);
    fprintf(fp, "%ld\n", pABite->biteSize);
    fprintf(fp, "%ld\n", pABite->biteRate);
    fprintf(fp, "%ld\n", pABite->data.data_len);
    for (i = 0; i < pABite->data.data_len; i++) {
        if (!(i % 8)) {
            fprintf(fp, "\n");
```

```
        }
        fprintf(fp, "%d ", pABite->data.data_val[i]);
    }
    fprintf(fp, "\n");
}


void
inputAudioBites(fp, pABites)
    FILE           *fp;
    audioBites     *pABites;
{
    int             i;

    fscanf(fp, "%d", &pABites->audioBites_len);
    pABites->audioBites_val = (audioBite *)
        malloc(pABites->audioBites_len * sizeof(audioBite));
    for (i = 0; i < pABites->audioBites_len; i++) {
        inputAudioBite(fp, &pABites->audioBites_val[i]);
    }
}

void
outputAudioBites(fp, pABites)
    FILE           *fp;
    audioBites     *pABites;
{
    int             i;

    fprintf(fp, "%d\n", pABites->audioBites_len);
    for (i = 0; i < pABites->audioBites_len; i++) {
        outputAudioBite(fp, &pABites->audioBites_val[i]);
    }

}

void
audioBiteXDRFree(pABite)
    audioBite      *pABite;
{
    xdr_free(xdr_audioBite, (char *) pABite);
    memset(pABite, 0, sizeof(audioBite));
}

void
audioBitesXDRFree(pABites)
    audioBites     *pABites;
{
    xdr_free(xdr_audioBites, (char *) pABites);
    memset(pABites, 0, sizeof(audioBites));
}

void
```

```
audioClipXDRFree(pAClip)
    audioClip     *pAClip;
{
    xdr_free(xdr_audioClip, (char *) pAClip);
    memset(pAClip, 0, sizeof(audioClip));
}


#ifdef STANDALONE
main(argc, argv)
#else               /* STANDALONE */
audioBiteMain(argc, argv)
#endif              /* STANDALONE */
    int             argc;
    char          **argv;
{
    static audioBite aBite;
    static audioBites aBites;
    audioBites     *cpABites;
    audioBite      *cpABite;

    switch (argc) {
    case 1:     /* receive aBite */
        audioBiteIn(0 /* stdin */ , &aBite);
        outputAudioBite(stdout, &aBite);
        cpABite = copyAudioBite(&aBite, NULL);
        outputAudioBite(stdout, cpABite);
        freeAudioBite(cpABite);

        break;
    case 2:     /* receive aBite */
        inputAudioBite(stdin, &aBite);
#ifdef DEBUG
        outputAudioBite(stderr, &aBite);
#endif
        audioBiteOut(1 /* stdout */ , &aBite);

        break;
    case 3:     /* receive aBites */
        audioBitesIn(0 /* stdin */ , &aBites);
        outputAudioBites(stdout, &aBites);
        cpABites = copyAudioBites(&aBites, NULL);
        outputAudioBites(stdout, cpABites);
        freeAudioBites(cpABites);

        break;
    case 4:     /* receive aBites */
        inputAudioBites(stdin, &aBites);
#ifdef DEBUG
        outputAudioBites(stderr, &aBites);
#endif
        audioBitesOut(1 /* stdout */ , &aBites);
```

```
            break;
        }

    }
```

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */
/*****************************************************************************
    ***/
/*****************************************************************************
    ***/
/**
    **/
/** This SHASTRA software is not in the Public Domain. It is distributed on
    **/
/** a person to person basis, solely for educational use and permission is
    **/
/** NOT granted for its transfer to anyone or for its use in any commercial
    **/
/** product.  There is NO warranty on the available software and neither
    **/
/** Purdue University nor the Applied Algebra and Geometry group directed
    **/
/** by C.  Bajaj accept responsibility for the consequences of its use.
    **/
/**
    **/
/*****************************************************************************
    ***/
/*****************************************************************************
    ***/
#include <rpc/rpc.h>
#include <shastra/datacomm/audioBite.h>

bool_t
xdr_audioBite(xdrs, objp)
    XDR *xdrs;
    audioBite *objp;
{
    if (!xdr_u_long(xdrs, &objp->lIdTag)) {
        return (FALSE);
    }
    if (!xdr_u_long(xdrs, &objp->lSIdTag)) {
        return (FALSE);
    }
    if (!xdr_u_long(xdrs, &objp->lPerms)) {
        return (FALSE);
    }
    if (!xdr_u_short(xdrs, &objp->biteFormat)) {
        return (FALSE);
    }
    if (!xdr_u_short(xdrs, &objp->biteComp)) {
        return (FALSE);
    }
    if (!xdr_u_long(xdrs, &objp->biteMode)) {
        return (FALSE);
```

```
        }
        if (!xdr_u_long(xdrs, &objp->biteSize)) {
            return (FALSE);
        }
        if (!xdr_u_long(xdrs, &objp->biteRate)) {
            return (FALSE);
        }
        if (!xdr_bytes(xdrs, (char **)&objp->data.data_val, (u_int *)&objp->
            data.data_len, ~0)) {
            return (FALSE);
        }
        return (TRUE);
    }

    bool_t
    xdr_audioBite_P(xdrs, objp)
        XDR *xdrs;
        audioBite_P *objp;
    {
        if (!xdr_pointer(xdrs, (char **)objp, sizeof(audioBite), xdr_audioBite)
            ) {
            return (FALSE);
        }
        return (TRUE);
    }

    bool_t
    xdr_audioBites(xdrs, objp)
        XDR *xdrs;
        audioBites *objp;
    {
        if (!xdr_array(xdrs, (char **)&objp->audioBites_val, (u_int *)&objp->
            audioBites_len, ~0, sizeof(audioBite), xdr_audioBite)) {
            return (FALSE);
        }
        return (TRUE);
    }

    bool_t
    xdr_audioBites_P(xdrs, objp)
        XDR *xdrs;
        audioBites_P *objp;
    {
        if (!xdr_pointer(xdrs, (char **)objp, sizeof(audioBites),
            xdr_audioBites)) {
            return (FALSE);
        }
        return (TRUE);
    }

    bool_t
    xdr_audioClip(xdrs, objp)
        XDR *xdrs;
```

```
    audioClip *objp;
{
    if (!xdr_vector(xdrs, (char *)objp->sbName, 32, sizeof(char), xdr_char)
        ) {
        return (FALSE);
    }
    if (!xdr_u_long(xdrs, &objp->lIdTag)) {
        return (FALSE);
    }
    if (!xdr_u_long(xdrs, &objp->lSIdTag)) {
        return (FALSE);
    }
    if (!xdr_u_long(xdrs, &objp->lPerms)) {
        return (FALSE);
    }
    if (!xdr_long(xdrs, &objp->lType)) {
        return (FALSE);
    }
    if (!xdr_long(xdrs, &objp->lPointer)) {
        return (FALSE);
    }
    if (!xdr_pointer(xdrs, (char **)&objp->pABites, sizeof(audioBites),
        xdr_audioBites)) {
        return (FALSE);
    }
    return (TRUE);
}
```

```
/*****************************************************************************
    ***/
/*****************************************************************************
    ***/
/**
    **/
/** This SHASTRA software is not in the Public Domain. It is distributed on
    **/
/** a person to person basis, solely for educational use and permission is
    **/
/** NOT granted for its transfer to anyone or for its use in any commercial
    **/
/** product.  There is NO warranty on the available software and neither
    **/
/** Purdue University nor the Applied Algebra and Geometry group directed
    **/
/** by C.  Bajaj accept responsibility for the consequences of its use.
    **/
/**
    **/
/*****************************************************************************
    ***/
/*****************************************************************************
    ***/
#include <shastra/datacomm/ipimage.h>


bool_t
xdr_ipimageData(xdrs, objp)
    XDR *xdrs;
    ipimageData *objp;
{
    if (!xdr_vector(xdrs, (char *)objp->sbName, 32, sizeof(char), xdr_char)
        ) {
        return (FALSE);
    }
    if (!xdr_u_long(xdrs, &objp->lIdTag)) {
        return (FALSE);
    }
    if (!xdr_u_long(xdrs, &objp->lSIdTag)) {
        return (FALSE);
    }
    if (!xdr_u_long(xdrs, &objp->lPerms)) {
        return (FALSE);
    }
    if (!xdr_vector(xdrs, (char *)objp->centroid, 3, sizeof(double),
        xdr_double)) {
        return (FALSE);
    }
    if (!xdr_int(xdrs, &objp->dispMode)) {
        return (FALSE);
    }
    if (!xdr_int(xdrs, &objp->color)) {
```

```
            return (FALSE);
        }
        if (!xdr_int(xdrs, &objp->shade)) {
            return (FALSE);
        }
        if (!xdr_int(xdrs, &objp->dispInfo)) {
            return (FALSE);
        }
        if (!xdr_pointer(xdrs, (char **)&objp->iPoly, sizeof(iPoly), xdr_iPoly)
            ) {
            return (FALSE);
        }
        return (TRUE);
    }
```

```
/*****************************************************************************
    ***/
/*****************************************************************************
    ***/
/**
    **/
/** This SHASTRA software is not in the Public Domain. It is distributed on
    **/
/** a person to person basis, solely for educational use and permission is
    **/
/** NOT granted for its transfer to anyone or for its use in any commercial
    **/
/** product.  There is NO warranty on the available software and neither
    **/
/** Purdue University nor the Applied Algebra and Geometry group directed
    **/
/** by C.  Bajaj accept responsibility for the consequences of its use.
    **/
/**
    **/
/*****************************************************************************
    ***/
/*****************************************************************************
    ***/
#include <stdio.h>
#include <ctype.h>
#include <malloc.h>

#include <shastra/network/mplex.h>
#include <shastra/network/rpc.h>
#include <ipoly/iPolyH.h>
#include <ipoly/ipolyutil.h>
#include <shastra/datacomm/ipimage.h>

void ipimageDataXDRFree(Prot1( ipimageData*));
bool_t xdr_ipimageData();
#define STANDALONEnn

#define DEBUGnn

int
ipimageDataOut(fd, pImage)
     int             fd;
     ipimageData      *pImage;
{
  XDR             xdrs;
  int       retVal = 0;

#ifdef STANDALONE
  {
    FILE           *fp;
    fp = stdout /* fdopen(fd,"w") */ ;
    xdrstdio_create(&xdrs, fp, XDR_ENCODE);
```

```
    if(!xdr_ipimageData(&xdrs, pImage)){
      retVal = -1;
    }
  }
#else               /* STANDALONE */
  /*
    xdrstdio_create(mplexXDRSEnc(fd), mplexOutStream(fd), XDR_ENCODE);
    */
  if(!xdr_ipimageData(mplexXDRSEnc(fd), pImage)){
    retVal = -1;
  }
#endif              /* STANDALONE */
  return retVal;
}

int
ipimageDataIn(fd, pImage)
     int             fd;
     ipimageData      *pImage;
{
  XDR             xdrs;
  int       retVal = 0;

  ipimageDataXDRFree(pImage);
#ifdef STANDALONE
  {
    FILE           *fp;
    fp = stdin /* fdopen(fd,"r") */ ;
    xdrstdio_create(&xdrs, fp, XDR_DECODE);
    if(!xdr_ipimageData(&xdrs, pImage)){
      retVal = -1;
    }
  }
#else               /* STANDALONE */
  /*
    xdrstdio_create(mplexXDRSDec(fd), mplexInStream(fd), XDR_DECODE);
    */
  if(!xdr_ipimageData(mplexXDRSDec(fd), pImage)){
    retVal = -1;
  }
#endif              /* STANDALONE */
  return retVal;
}

int
ipimageDataMemOut(buf, size, pImage)
     char *buf;
     int size;
     ipimageData      *pImage;
{
  XDR             xdrs;
  int       retVal = 0;
```

```
  xdrmem_create(&xdrs, buf, size, XDR_ENCODE);
  if(!xdr_ipimageData(&xdrs, pImage)){
    retVal = -1;
  }
  xdr_destroy(&xdrs);
  return retVal;
}

int
ipimageDataMemIn(buf, size, pImage)
     char *buf;
     int size;
     ipimageData      *pImage;
{
  XDR             xdrs;
  int       retVal = 0;

  ipimageDataXDRFree(pImage);
  xdrmem_create(&xdrs, buf, size, XDR_DECODE);
  if(!xdr_ipimageData(&xdrs, pImage)){
    retVal = -1;
  }
  xdr_destroy(&xdrs);
  return retVal;
}

void
ipimageDataXDRFree(pImage)
     ipimageData      *pImage;
{
  xdr_free(xdr_ipimageData, (char *) pImage);
  memset(pImage, 0, sizeof(ipimageData));
}

int
IPolyOut(fd, pIPoly)
     int             fd;
     iPoly      *pIPoly;
{
  XDR             xdrs;
  int             retVal = 0;

#ifdef STANDALONE
  {
    FILE           *fp;
    fp = stdout /* fdopen(fd,"w") */ ;
    xdrstdio_create(&xdrs, fp, XDR_ENCODE);
    if (!xdr_iPoly(&xdrs, pIPoly)) {
      retVal = -1;
    }
  }
#else               /* STANDALONE */
  if (!xdr_iPoly(mplexXDRSEnc(fd), pIPoly)) {
```

```
    retVal = -1;
  }
#endif              /* STANDALONE */
  return retVal;
}

int
IPolyIn(fd, pIPoly)
     int             fd;
     iPoly      *pIPoly;
{
  XDR             xdrs;
  int             retVal = 0;

  IPolyXDRFree(pIPoly);
#ifdef STANDALONE
  {
    FILE           *fp;
    fp = stdin /* fdopen(fd,"r") */ ;
    xdrstdio_create(&xdrs, fp, XDR_DECODE);
    if (!xdr_iPoly(&xdrs, pIPoly)) {
      retVal = -1;
    }
  }
#else               /* STANDALONE */
  if (!xdr_iPoly(mplexXDRSDec(fd), pIPoly)) {
    retVal = -1;
  }
#endif              /* STANDALONE */
  return retVal;
}

int
IPolyMemOut(buf, size, pIPoly)
     char *buf;
     int size;
     iPoly      *pIPoly;
{
  XDR             xdrs;
  int     retVal = 0;

  xdrmem_create(&xdrs, buf, size, XDR_ENCODE);
  if(!xdr_iPoly(&xdrs, pIPoly)){
    retVal = -1;
  }
  xdr_destroy(&xdrs);
  return retVal;
}

int
IPolyMemIn(buf, size, pIPoly)
     char *buf;
     int size;
```

```
    iPoly      *pIPoly;
{
  XDR             xdrs;
  int     retVal = 0;

  xdrmem_create(&xdrs, buf, size, XDR_DECODE);
  if(!xdr_iPoly(&xdrs, pIPoly)){
    retVal = -1;
  }
  xdr_destroy(&xdrs);
  return retVal;
}


void
inputIPoly(fp, pIPoly)
     FILE           *fp;
     iPoly      *pIPoly;
{
  int i,j;

  IPolyXDRFree(pIPoly);

  fscanf(fp,"%d", &IPolyNVerts(pIPoly));
  if(IPolyNVerts(pIPoly) > 0){
    IPolyVerts(pIPoly) = (iPolyPoint*)malloc(IPolyNVerts(pIPoly)*
                         sizeof(iPolyPoint));
  }
  for (i = 0; i < IPolyNVerts(pIPoly); i++) {
    fscanf(fp,"%lf%lf%lf", &IPolyVert(pIPoly,i)[0],
        &IPolyVert(pIPoly,i)[1], &IPolyVert(pIPoly,i)[2]);
  }
  fscanf(fp,"%d", &IPolyNVertFaceAdjs(pIPoly));
  if(IPolyNVertFaceAdjs(pIPoly) > 0){
    IPolyVertAdjFaces(pIPoly) = (iFaces *)malloc(IPolyNVertFaceAdjs(pIPoly)
        *
                         sizeof(iFaces));
  }
  for (i = 0; i < IPolyNVertFaceAdjs(pIPoly); i++) {
    fscanf(fp,"%d", &IPolyVertNFaceAdj(pIPoly, i));
    if(IPolyVertNFaceAdj(pIPoly, i) > 0){
      IPolyVertFaceAdjs(pIPoly, i) = (int *)malloc(IPolyVertNFaceAdj(pIPoly
          , i)
                           * sizeof(int));
    }
    for (j = 0; j < IPolyVertNFaceAdj(pIPoly, i); j++) {
      fscanf(fp, "%d", &IPolyVertFaceAdj(pIPoly, i, j));
    }
  }
  fscanf(fp,"%d", &IPolyNVertEdgeAdjs(pIPoly));
  if(IPolyNVertEdgeAdjs(pIPoly) > 0){
    IPolyVertAdjEdges(pIPoly) = (iEdges *)malloc(IPolyNVertEdgeAdjs(pIPoly)
        *
```

```
                        sizeof(iEdges));
  }
  for (i = 0; i < IPolyNVertEdgeAdjs(pIPoly); i++) {
    fscanf(fp,"%d", &IPolyVertNEdgeAdj(pIPoly, i));
    if(IPolyVertNEdgeAdj(pIPoly, i) > 0){
      IPolyVertEdgeAdjs(pIPoly, i) = (int *)malloc(IPolyVertNEdgeAdj(pIPoly
          , i)
                          * sizeof(int));
    }
    for (j = 0; j < IPolyVertNEdgeAdj(pIPoly, i); j++) {
      fscanf(fp, "%d", &IPolyVertEdgeAdj(pIPoly, i, j));
    }
  }
  fscanf(fp,"%d", &IPolyNVertNorms(pIPoly));
  if(IPolyNVertNorms(pIPoly) > 0){
    IPolyVertNorms(pIPoly) = (iPolyNormal*)malloc(IPolyNVertNorms(pIPoly)*
                          sizeof(iPolyNormal));
  }
  for (i = 0; i < IPolyNVertNorms(pIPoly); i++) {
    fscanf(fp,"%f%f%f", &IPolyVertNorm(pIPoly,i)[0],
        &IPolyVertNorm(pIPoly,i)[1], &IPolyVertNorm(pIPoly,i)[2]);
  }
  fscanf(fp,"%d", &IPolyNVertSizes(pIPoly));
  if(IPolyNVertSizes(pIPoly) > 0){
    IPolyVertSizes(pIPoly) = (iPolySize *)malloc(IPolyNVertSizes(pIPoly)*
                          sizeof(iPolySize));
  }
  for (i = 0; i < IPolyNVertSizes(pIPoly); i++) {
    fscanf(fp,"%lf", &IPolyVertSize(pIPoly,i));
  }
  switch(IPolyVertColorCode(pIPoly)){
  case ColorIndex:
    fscanf(fp,"%d", &IPolyNVertColors(pIPoly));
    if(IPolyNVertColors(pIPoly) > 0){
      IPolyVertColorArr(pIPoly) = (int *)malloc(IPolyNVertColors(pIPoly) *
                        sizeof(int));
    }
    for (i = 0; i < IPolyNVertColors(pIPoly); i++) {
      fscanf(fp,"%d", &IPolyVertColor(pIPoly,i));
    }
    break;
  case ShadeIndex:
    fscanf(fp,"%d", &IPolyNVertShades(pIPoly));
    if(IPolyNVertShades(pIPoly) > 0){
      IPolyVertShadeArr(pIPoly) = (int *)malloc(IPolyNVertShades(pIPoly) *
                        sizeof(int));
    }
    for (i = 0; i < IPolyNVertShades(pIPoly); i++) {
      fscanf(fp,"%d", &IPolyVertShade(pIPoly,i));
    }
    break;
  case ColorMapVal:
    fscanf(fp,"%d", &IPolyNVertValues(pIPoly));
```

```
  if(IPolyNVertValues(pIPoly) > 0){
    IPolyVertValueArr(pIPoly) = (float *)malloc(IPolyNVertValues(pIPoly)
        *
                    sizeof(float));
  }
  for (i = 0; i < IPolyNVertValues(pIPoly); i++) {
    fscanf(fp,"%f", &IPolyVertValue(pIPoly,i));
  }
  break;
case NoColor:
default:
  break;
}

fscanf(fp,"%d", &IPolyNEdges(pIPoly));
if(IPolyNEdges(pIPoly) > 0){
  IPolyEdges(pIPoly) = (iPolyEdgeVerts*)malloc(IPolyNEdges(pIPoly)*
                        sizeof(iPolyEdgeVerts));
}
for (i = 0; i < IPolyNEdges(pIPoly); i++) {
  fscanf(fp,"%d%d", &IPolyEdgeV1(pIPoly, i), &IPolyEdgeV2(pIPoly, i));
}
fscanf(fp,"%d", &IPolyNEdgeFaceAdjs(pIPoly));
if(IPolyNEdgeFaceAdjs(pIPoly) > 0){
  IPolyEdgeAdjFaces(pIPoly) = (iFaces *)malloc(IPolyNEdgeFaceAdjs(pIPoly)
      *
                        sizeof(iFaces));
}
for (i = 0; i < IPolyNEdgeFaceAdjs(pIPoly); i++) {
  fscanf(fp,"%d", &IPolyEdgeNFaceAdj(pIPoly, i));
  if(IPolyEdgeNFaceAdj(pIPoly, i) > 0){
    IPolyEdgeFaceAdjs(pIPoly, i) = (int *)malloc(IPolyEdgeNFaceAdj(pIPoly
        , i)
                          * sizeof(int));
  }
  for (j = 0; j < IPolyEdgeNFaceAdj(pIPoly, i); j++) {
    fscanf(fp, "%d", &IPolyEdgeFaceAdj(pIPoly, i, j));
  }
}
fscanf(fp,"%d", &IPolyNEdgeSizes(pIPoly));
if(IPolyNEdgeSizes(pIPoly) > 0){
  IPolyEdgeSizes(pIPoly) = (iPolySize*)malloc(IPolyNEdgeSizes(pIPoly)*
                         sizeof(iPolySize));
}
for (i = 0; i < IPolyNEdgeSizes(pIPoly); i++) {
  fscanf(fp,"%lf", &IPolyEdgeSize(pIPoly,i));
}
switch(IPolyEdgeColorCode(pIPoly)){
case ColorIndex:
  fscanf(fp,"%d", &IPolyNEdgeColors(pIPoly));
  if(IPolyNEdgeColors(pIPoly) > 0){
    IPolyEdgeColorArr(pIPoly) = (int *)malloc(IPolyNEdgeColors(pIPoly) *
                      sizeof(int));
```

```
  }
  for (i = 0; i < IPolyNEdgeColors(pIPoly); i++) {
    fscanf(fp,"%d", &IPolyEdgeColor(pIPoly,i));
  }
  break;
case ShadeIndex:
  fscanf(fp,"%d", &IPolyNEdgeShades(pIPoly));
  if(IPolyNEdgeShades(pIPoly) > 0){
    IPolyEdgeShadeArr(pIPoly) = (int *)malloc(IPolyNEdgeShades(pIPoly) *
                        sizeof(int));
  }
  for (i = 0; i < IPolyNEdgeShades(pIPoly); i++) {
    fscanf(fp,"%d", &IPolyEdgeShade(pIPoly,i));
  }
  break;
case ColorMapVal:
  fscanf(fp,"%d", &IPolyNEdgeValues(pIPoly));
  if(IPolyNEdgeValues(pIPoly) > 0){
    IPolyEdgeValueArr(pIPoly) = (float *)malloc(IPolyNEdgeValues(pIPoly)
        *
                    sizeof(float));
  }
  for (i = 0; i < IPolyNEdgeValues(pIPoly); i++) {
    fscanf(fp,"%f", &IPolyEdgeValue(pIPoly,i));
  }
  break;
case NoColor:
default:
  break;
}

fscanf(fp,"%d", &IPolyNEdgeFaces(pIPoly));
if(IPolyNEdgeFaces(pIPoly) > 0){
  IPolyEdgeFaces(pIPoly) = (iEdges*)malloc(IPolyNEdgeFaces(pIPoly)*
                         sizeof(iEdges));
}
for (i = 0; i < IPolyNEdgeFaces(pIPoly); i++) {
  fscanf(fp,"%d", &IPolyNFaceEdges(pIPoly, i));
  if(IPolyNFaceEdges(pIPoly, i) > 0){
    IPolyFaceEdges(pIPoly, i) = (int *)malloc(IPolyNFaceEdges(pIPoly, i)*
                        sizeof(int));
  }
  for (j = 0; j < IPolyNFaceEdges(pIPoly, i); j++) {
    fscanf(fp, "%d", &IPolyFaceEdge(pIPoly, i, j));
  }
}
fscanf(fp,"%d", &IPolyNVertFaces(pIPoly));
if(IPolyNVertFaces(pIPoly) > 0){
  IPolyVertFaces(pIPoly) = (iVerts*)malloc(IPolyNVertFaces(pIPoly)*
                         sizeof(iVerts));
}
for (i = 0; i < IPolyNVertFaces(pIPoly); i++) {
  fscanf(fp,"%d", &IPolyNFaceVerts(pIPoly, i));
```

```
  if(IPolyNFaceVerts(pIPoly, i) > 0){
    IPolyFaceVerts(pIPoly, i) = (int *)malloc(IPolyNFaceVerts(pIPoly, i)*
                          sizeof(int));
  }
  for (j = 0; j < IPolyNFaceVerts(pIPoly, i); j++) {
    fscanf(fp, "%d", &IPolyFaceVert(pIPoly, i, j));
  }
}
fscanf(fp,"%d", &IPolyNFaceSizes(pIPoly));
if(IPolyNFaceSizes(pIPoly) > 0){
  IPolyFaceSizes(pIPoly) = (iPolySize*)malloc(IPolyNFaceSizes(pIPoly)*
                          sizeof(iPolySize));
}
for (i = 0; i < IPolyNFaceSizes(pIPoly); i++) {
  fscanf(fp,"%lf", &IPolyFaceSize(pIPoly,i));
}
fscanf(fp,"%d", &IPolyNFaceNorms(pIPoly));
if(IPolyNFaceNorms(pIPoly) > 0){
  IPolyFaceNorms(pIPoly) = (iPolyNormal*)malloc(IPolyNFaceNorms(pIPoly)*
                           sizeof(iPolyNormal));
}
for (i = 0; i < IPolyNFaceNorms(pIPoly); i++) {
  fscanf(fp,"%f%f%f", &IPolyFaceNorm(pIPoly,i)[0],
      &IPolyFaceNorm(pIPoly,i)[1], &IPolyFaceNorm(pIPoly,i)[2]);
}
switch(IPolyFaceColorCode(pIPoly)){
case ColorIndex:
  fscanf(fp,"%d", &IPolyNFaceColors(pIPoly));
  IPolyFaceColorArr(pIPoly) = (int *)malloc(IPolyNFaceColors(pIPoly) *
                     sizeof(int));
  for (i = 0; i < IPolyNFaceColors(pIPoly); i++) {
    fscanf(fp,"%d", &IPolyFaceColor(pIPoly,i));
  }
  break;
case ShadeIndex:
  fscanf(fp,"%d", &IPolyNFaceShades(pIPoly));
  if(IPolyNFaceShades(pIPoly) > 0){
    IPolyFaceShadeArr(pIPoly) = (int *)malloc(IPolyNFaceShades(pIPoly) *
                        sizeof(int));
  }
  for (i = 0; i < IPolyNFaceShades(pIPoly); i++) {
    fscanf(fp,"%d", &IPolyFaceShade(pIPoly,i));
  }
  break;
case ColorMapVal:
  fscanf(fp,"%d", &IPolyNFaceValues(pIPoly));
  if(IPolyNFaceValues(pIPoly) > 0){
    IPolyFaceValueArr(pIPoly) = (float *)malloc(IPolyNFaceValues(pIPoly)
        *
                     sizeof(float));
  }
  for (i = 0; i < IPolyNFaceValues(pIPoly); i++) {
    fscanf(fp,"%f", &IPolyFaceValue(pIPoly,i));
```

```
  }
  break;
case NoColor:
default:
  break;
}

fscanf(fp,"%d", &IPolyNElmts(pIPoly));
if(IPolyNElmts(pIPoly) > 0){
  IPolyElmts(pIPoly) = (iFaces*)malloc(IPolyNElmts(pIPoly)*
                       sizeof(iFaces));
}
for (i = 0; i < IPolyNElmts(pIPoly); i++) {
  fscanf(fp,"%d", &IPolyNElmtFaces(pIPoly, i));
  if(IPolyNElmtFaces(pIPoly, i) > 0){
    IPolyElmtFaces(pIPoly, i) = (int *)malloc(IPolyNElmtFaces(pIPoly, i)*
                      sizeof(int));
  }
  for (j = 0; j < IPolyNElmtFaces(pIPoly, i); j++) {
    fscanf(fp, "%d", &IPolyElmtFace(pIPoly, i, j));
  }
}
fscanf(fp,"%d", &IPolyNElmtSizes(pIPoly));
if(IPolyNElmtSizes(pIPoly) > 0){
  IPolyElmtSizes(pIPoly) = (iPolySize*)malloc(IPolyNElmtSizes(pIPoly)*
                      sizeof(iPolySize));
}
for (i = 0; i < IPolyNElmtSizes(pIPoly); i++) {
  fscanf(fp,"%lf", &IPolyElmtSize(pIPoly,i));
}
switch(IPolyElmtColorCode(pIPoly)){
case ColorIndex:
  fscanf(fp,"%d", &IPolyNElmtColors(pIPoly));
  IPolyElmtColorArr(pIPoly) = (int *)malloc(IPolyNElmtColors(pIPoly) *
                    sizeof(int));
  for (i = 0; i < IPolyNElmtColors(pIPoly); i++) {
    fscanf(fp,"%d", &IPolyElmtColor(pIPoly,i));
  }
  break;
case ShadeIndex:
  fscanf(fp,"%d", &IPolyNElmtShades(pIPoly));
  if(IPolyNElmtShades(pIPoly) > 0){
    IPolyElmtShadeArr(pIPoly) = (int *)malloc(IPolyNElmtShades(pIPoly) *
                      sizeof(int));
  }
  for (i = 0; i < IPolyNElmtShades(pIPoly); i++) {
    fscanf(fp,"%d", &IPolyElmtShade(pIPoly,i));
  }
  break;
case ColorMapVal:
  fscanf(fp,"%d", &IPolyNElmtValues(pIPoly));
  if(IPolyNElmtValues(pIPoly) > 0){
    IPolyElmtValueArr(pIPoly) = (float *)malloc(IPolyNElmtValues(pIPoly)
```

```
        *
                  sizeof(float));
  }
  for (i = 0; i < IPolyNElmtValues(pIPoly); i++) {
    fscanf(fp,"%f", &IPolyElmtValue(pIPoly,i));
  }
  break;
case NoColor:
default:
  break;
}

fscanf(fp,"%d", &IPolyNEGroups(pIPoly));
if(IPolyNEGroups(pIPoly) > 0){
  IPolyEGroups(pIPoly) = (iRanges *)malloc(IPolyNEGroups(pIPoly)*
                         sizeof(iRanges));
}
for (i = 0; i < IPolyNEGroups(pIPoly); i++) {
  fscanf(fp,"%d", &IPolyEGroupLen(pIPoly, i));
  if(IPolyEGroupLen(pIPoly, i) > 0){
    IPolyEGroupVal(pIPoly, i) = (range *)malloc(IPolyEGroupLen(pIPoly, i)
        *
                          sizeof(range));
  }
  for (j = 0; j < IPolyEGroupLen(pIPoly, i); j++){
    fscanf(fp, "%d%d", &IPolyEGroupLow(pIPoly, i, j),
       &IPolyEGroupHigh(pIPoly, i, j));
  }
}

fscanf(fp,"%d", &IPolyNFGroups(pIPoly));
if(IPolyNFGroups(pIPoly) > 0){
  IPolyFGroups(pIPoly) = (iRanges *)malloc(IPolyNFGroups(pIPoly)*
                         sizeof(iRanges));
}
for (i = 0; i < IPolyNFGroups(pIPoly); i++) {
  fscanf(fp,"%d", &IPolyFGroupLen(pIPoly, i));
  if(IPolyFGroupLen(pIPoly, i) > 0){
    IPolyFGroupVal(pIPoly, i) = (range *)malloc(IPolyFGroupLen(pIPoly, i)
        *
                          sizeof(range));
  }
  for (j = 0; j < IPolyFGroupLen(pIPoly, i); j++){
    fscanf(fp, "%d%d", &IPolyFGroupLow(pIPoly, i, j),
       &IPolyFGroupHigh(pIPoly, i, j));
  }
}

fscanf(fp,"%d", &IPolyNColors(pIPoly));
if(IPolyNColors(pIPoly) > 0){
  IPolyColors(pIPoly) = (iPolyRGB*)malloc(IPolyNColors(pIPoly)*
                        sizeof(iPolyRGB));
}
```

```
  for (i = 0; i < IPolyNColors(pIPoly); i++) {
    fscanf(fp, "%f%f%f", &IPolyColor(pIPoly,i)[0],
        &IPolyColor(pIPoly,i)[1], &IPolyColor(pIPoly,i)[2]);
  }

}

void
outputIPoly(fp, pIPoly)
     FILE *fp;
     iPoly *pIPoly;
{
  int i,j;

  fprintf(fp,"/*IPoly Vertex Coordinates*/\n");
  fprintf(fp,"%d\n", IPolyNVerts(pIPoly));
  for (i = 0; i < IPolyNVerts(pIPoly); i++) {
    fprintf(fp,"%lf %lf %lf\n", IPolyVert(pIPoly,i)[0],
        IPolyVert(pIPoly,i)[1], IPolyVert(pIPoly,i)[2]);
  }
  fprintf(fp,"/*IPoly Vertex Face Adjacencies*/\n");
  fprintf(fp,"%d\n", IPolyNVertFaceAdjs(pIPoly));
  for (i = 0; i < IPolyNVertFaceAdjs(pIPoly); i++) {
    fprintf(fp,"%d ", IPolyVertNFaceAdj(pIPoly, i));
    for (j = 0; j < IPolyVertNFaceAdj(pIPoly, i); j++) {
      fprintf(fp, "%d ", IPolyVertFaceAdj(pIPoly, i, j));
    }
    fprintf(fp, "\n");
  }
  fprintf(fp,"/*IPoly Vertex Edge Adjacencies*/\n");
  fprintf(fp,"%d\n", IPolyNVertEdgeAdjs(pIPoly));
  for (i = 0; i < IPolyNVertEdgeAdjs(pIPoly); i++) {
    fprintf(fp,"%d ", IPolyVertNEdgeAdj(pIPoly, i));
    for (j = 0; j < IPolyVertNEdgeAdj(pIPoly, i); j++) {
      fprintf(fp, "%d ", IPolyVertEdgeAdj(pIPoly, i, j));
    }
    fprintf(fp, "\n");
  }
  fprintf(fp,"/*IPoly Vertex Normals*/\n");
  fprintf(fp,"%d\n", IPolyNVertNorms(pIPoly));
  for (i = 0; i < IPolyNVertNorms(pIPoly); i++) {
    fprintf(fp,"%f %f %f\n", IPolyVertNorm(pIPoly,i)[0],
        IPolyVertNorm(pIPoly,i)[1], IPolyVertNorm(pIPoly,i)[2]);
  }
  fprintf(fp,"/*IPoly Vertex Sizes*/\n");
  fprintf(fp,"%d\n", IPolyNVertSizes(pIPoly));
  for (i = 0; i < IPolyNVertSizes(pIPoly); i++) {
    fprintf(fp,"%lf\n", IPolyVertSize(pIPoly,i));
  }
  switch(IPolyVertColorCode(pIPoly)){
  case ColorIndex:
    fprintf(fp,"/*IPoly Vertex Colors*/\n");
    fprintf(fp,"%d\n", IPolyNVertColors(pIPoly));
```

```
  for (i = 0; i < IPolyNVertColors(pIPoly); i++) {
    fprintf(fp,"%d\n", IPolyVertColor(pIPoly,i));
  }
  break;
case ShadeIndex:
  fprintf(fp,"/*IPoly Vertex Shades*/\n");
  fprintf(fp,"%d\n", IPolyNVertShades(pIPoly));
  for (i = 0; i < IPolyNVertShades(pIPoly); i++) {
    fprintf(fp,"%d\n", IPolyVertShade(pIPoly,i));
  }
  break;
case ColorMapVal:
  fprintf(fp,"/*IPoly Vertex Color Values*/\n");
  fprintf(fp,"%d\n", IPolyNVertValues(pIPoly));
  for (i = 0; i < IPolyNVertValues(pIPoly); i++) {
    fprintf(fp,"%f\n", IPolyVertValue(pIPoly,i));
  }
  break;
case NoColor:
default:
  break;
}

fprintf(fp,"/*IPoly Edge Vertex Pairs*/\n");
fprintf(fp,"%d\n", IPolyNEdges(pIPoly));
for (i = 0; i < IPolyNEdges(pIPoly); i++) {
  fprintf(fp,"%d %d\n", IPolyEdgeV1(pIPoly, i), IPolyEdgeV2(pIPoly, i));
}
fprintf(fp,"/*IPoly Edge Face Adjacencies*/\n");
fprintf(fp,"%d\n", IPolyNEdgeFaceAdjs(pIPoly));
for (i = 0; i < IPolyNEdgeFaceAdjs(pIPoly); i++) {
  fprintf(fp,"%d ", IPolyEdgeNFaceAdj(pIPoly, i));
  for (j = 0; j < IPolyEdgeNFaceAdj(pIPoly, i); j++) {
    fprintf(fp, "%d ", IPolyEdgeFaceAdj(pIPoly, i, j));
  }
  fprintf(fp, "\n");
}
fprintf(fp,"/*IPoly Edge Sizes*/\n");
fprintf(fp,"%d\n", IPolyNEdgeSizes(pIPoly));
for (i = 0; i < IPolyNEdgeSizes(pIPoly); i++) {
  fprintf(fp,"%lf\n", IPolyEdgeSize(pIPoly,i));
}
switch(IPolyEdgeColorCode(pIPoly)){
case ColorIndex:
  fprintf(fp,"/*IPoly Edge Colors*/\n");
  fprintf(fp,"%d\n", IPolyNEdgeColors(pIPoly));
  for (i = 0; i < IPolyNEdgeColors(pIPoly); i++) {
    fprintf(fp,"%d\n", IPolyEdgeColor(pIPoly,i));
  }
  break;
case ShadeIndex:
  fprintf(fp,"/*IPoly Edge Shades*/\n");
  fprintf(fp,"%d\n", IPolyNEdgeShades(pIPoly));
```

```
  for (i = 0; i < IPolyNEdgeShades(pIPoly); i++) {
    fprintf(fp,"%d\n", IPolyEdgeShade(pIPoly,i));
  }
  break;
case ColorMapVal:
  fprintf(fp,"/*IPoly Edge Color Values*/\n");
  fprintf(fp,"%d\n", IPolyNEdgeValues(pIPoly));
  for (i = 0; i < IPolyNEdgeValues(pIPoly); i++) {
    fprintf(fp,"%f\n", IPolyEdgeValue(pIPoly,i));
  }
  break;
case NoColor:
default:
  break;
}

fprintf(fp,"/*IPoly Faces by Edge*/\n");
fprintf(fp,"%d\n", IPolyNEdgeFaces(pIPoly));
for (i = 0; i < IPolyNEdgeFaces(pIPoly); i++) {
  fprintf(fp,"%d ", IPolyNFaceEdges(pIPoly, i));
  for (j = 0; j < IPolyNFaceEdges(pIPoly, i); j++) {
    fprintf(fp, "%d ", IPolyFaceEdge(pIPoly, i, j));
  }
  fprintf(fp, "\n");
}
fprintf(fp,"/*IPoly Faces by Vertex*/\n");
fprintf(fp,"%d\n", IPolyNVertFaces(pIPoly));
for (i = 0; i < IPolyNVertFaces(pIPoly); i++) {
  fprintf(fp,"%d ", IPolyNFaceVerts(pIPoly, i));
  for (j = 0; j < IPolyNFaceVerts(pIPoly, i); j++) {
    fprintf(fp, "%d ", IPolyFaceVert(pIPoly, i, j));
  }
  fprintf(fp, "\n");
}
fprintf(fp,"/*IPoly Face Sizes*/\n");
fprintf(fp,"%d\n", IPolyNFaceSizes(pIPoly));
for (i = 0; i < IPolyNFaceSizes(pIPoly); i++) {
  fprintf(fp,"%lf\n", IPolyFaceSize(pIPoly,i));
}
fprintf(fp,"/*IPoly Face Normals*/\n");
fprintf(fp,"%d\n", IPolyNFaceNorms(pIPoly));
for (i = 0; i < IPolyNFaceNorms(pIPoly); i++) {
  fprintf(fp,"%f %f %f\n", IPolyFaceNorm(pIPoly,i)[0],
      IPolyFaceNorm(pIPoly,i)[1], IPolyFaceNorm(pIPoly,i)[2]);
}
switch(IPolyFaceColorCode(pIPoly)){
case ColorIndex:
  fprintf(fp,"/*IPoly Face Colors*/\n");
  fprintf(fp,"%d\n", IPolyNFaceColors(pIPoly));
  for (i = 0; i < IPolyNFaceColors(pIPoly); i++) {
    fprintf(fp,"%d\n", IPolyFaceColor(pIPoly,i));
  }
  break;
```

```
case ShadeIndex:
  fprintf(fp,"/*IPoly Face Shades*/\n");
  fprintf(fp,"%d\n", IPolyNFaceShades(pIPoly));
  for (i = 0; i < IPolyNFaceShades(pIPoly); i++) {
    fprintf(fp,"%d\n", IPolyFaceShade(pIPoly,i));
  }
  break;
case ColorMapVal:
  fprintf(fp,"/*IPoly Face Color Values*/\n");
  fprintf(fp,"%d\n", IPolyNFaceValues(pIPoly));
  for (i = 0; i < IPolyNFaceValues(pIPoly); i++) {
    fprintf(fp,"%f\n", IPolyFaceValue(pIPoly,i));
  }
  break;
case NoColor:
default:
  break;
}

fprintf(fp,"/*IPoly Elements*/\n");
fprintf(fp,"%d\n", IPolyNElmts(pIPoly));
for (i = 0; i < IPolyNElmts(pIPoly); i++) {
  fprintf(fp,"%d\n", IPolyNElmtFaces(pIPoly, i));
  for (j = 0; j < IPolyNElmtFaces(pIPoly, i); j++) {
    fprintf(fp, "%d ", IPolyElmtFace(pIPoly, i, j));
  }
  fprintf(fp, "\n");
}
fprintf(fp,"/*IPoly Element Sizes*/\n");
fprintf(fp,"%d\n", IPolyNElmtSizes(pIPoly));
for (i = 0; i < IPolyNElmtSizes(pIPoly); i++) {
  fprintf(fp,"%lf\n", &IPolyElmtSize(pIPoly,i));
}
switch(IPolyElmtColorCode(pIPoly)){
case ColorIndex:
  fprintf(fp,"/*IPoly Element Colors*/\n");
  fprintf(fp,"%d\n", IPolyNElmtColors(pIPoly));
  for (i = 0; i < IPolyNElmtColors(pIPoly); i++) {
    fprintf(fp,"%d\n", IPolyElmtColor(pIPoly,i));
  }
  break;
case ShadeIndex:
  fprintf(fp,"/*IPoly Element Shades*/\n");
  fprintf(fp,"%d\n", IPolyNElmtShades(pIPoly));
  for (i = 0; i < IPolyNElmtShades(pIPoly); i++) {
    fprintf(fp,"%d\n", IPolyElmtShade(pIPoly,i));
  }
  break;
case ColorMapVal:
  fprintf(fp,"/*IPoly Element Values*/\n");
  fprintf(fp,"%d\n", IPolyNElmtValues(pIPoly));
  for (i = 0; i < IPolyNElmtValues(pIPoly); i++) {
    fprintf(fp,"%f\n", IPolyElmtValue(pIPoly,i));
```

```
    }
    break;
  case NoColor:
  default:
    break;
  }

  fprintf(fp,"/*IPoly Edge Groups*/\n");
  fprintf(fp,"%d\n", IPolyNEGroups(pIPoly));
  for (i = 0; i < IPolyNEGroups(pIPoly); i++) {
    fprintf(fp,"%d\n", IPolyEGroupLen(pIPoly, i));
    for (j = 0; j < IPolyEGroupLen(pIPoly, i); j++){
      fprintf(fp, "%d %d\n",
          IPolyEGroupLow(pIPoly, i, j), IPolyEGroupHigh(pIPoly, i, j));
    }
  }

  fprintf(fp,"/*IPoly Face Groups*/\n");
  fprintf(fp,"%d\n", IPolyNFGroups(pIPoly));
  for (i = 0; i < IPolyNFGroups(pIPoly); i++) {
    fprintf(fp,"%d\n", IPolyFGroupLen(pIPoly, i));
    for (j = 0; j < IPolyFGroupLen(pIPoly, i); j++){
      fprintf(fp, "%d %d\n",
          IPolyFGroupLow(pIPoly, i, j), IPolyFGroupHigh(pIPoly, i, j));
    }
  }

  fprintf(fp,"/*IPoly Colors*/\n");
  fprintf(fp,"%d\n", IPolyNColors(pIPoly));
  for (i = 0; i < IPolyNColors(pIPoly); i++) {
    fprintf(fp, "%f %f %f\n", IPolyColor(pIPoly,i)[0],
        IPolyColor(pIPoly,i)[1], IPolyColor(pIPoly,i)[2]);
  }

}

void
freeIPoly(pIPoly)
     iPoly      *pIPoly;
{
  IPolyXDRFree(pIPoly);
}

iPoly      *
copyIPoly(pIPoly, destpIPoly)
     iPoly      *pIPoly;
     iPoly      *destpIPoly;
{
  char *buf;
  int bufSize = 65536;
  iPoly *newIPoly;

  buf = malloc(bufSize);
```

```
  while(IPolyMemOut(buf, bufSize, pIPoly)== -1){
    bufSize *=2;
    buf = realloc(buf, bufSize);
  }
  if(destpIPoly){
    newIPoly = destpIPoly;
  }
  else{
    newIPoly = (iPoly *)malloc(sizeof(iPoly));
    memset(newIPoly, 0, sizeof(iPoly));
  }
  IPolyMemIn(buf, bufSize, newIPoly);
  free(buf);

  return newIPoly;
}

void
IPolyXDRFree(pIPoly)
     iPoly      *pIPoly;
{
  xdr_free(xdr_iPoly, (char *) pIPoly);
  memset(pIPoly, 0, sizeof(iPoly));
}


int
IPolysOut(fd, pIPolys)
     int             fd;
     iPolys      *pIPolys;
{
  XDR             xdrs;
  int             retVal = 0;

#ifdef STANDALONE
  {
    FILE           *fp;
    fp = stdout /* fdopen(fd,"w") */ ;
    xdrstdio_create(&xdrs, fp, XDR_ENCODE);
    if (!xdr_iPolys(&xdrs, pIPolys)) {
      retVal = -1;
    }
  }
#else               /* STANDALONE */
  if (!xdr_iPolys(mplexXDRSEnc(fd), pIPolys)) {
    retVal = -1;
  }
#endif              /* STANDALONE */
  return retVal;
}

int
```

```
IPolysIn(fd, pIPolys)
     int             fd;
     iPolys      *pIPolys;
{
  XDR             xdrs;
  int             retVal = 0;

  IPolysXDRFree(pIPolys);
#ifdef STANDALONE
  {
    FILE           *fp;
    fp = stdin /* fdopen(fd,"r") */ ;
    xdrstdio_create(&xdrs, fp, XDR_DECODE);
    if (!xdr_iPolys(&xdrs, pIPolys)) {
      retVal = -1;
    }
  }
#else               /* STANDALONE */
  if (!xdr_iPolys(mplexXDRSDec(fd), pIPolys)) {
    retVal = -1;
  }
#endif              /* STANDALONE */
  return retVal;
}


int
IPolysMemOut(buf, size, pIPolys)
     char *buf;
     int size;
     iPolys      *pIPolys;
{
  XDR             xdrs;
  int     retVal = 0;

  xdrmem_create(&xdrs, buf, size, XDR_ENCODE);
  if(!xdr_iPolys(&xdrs, pIPolys)){
    retVal = -1;
  }
  xdr_destroy(&xdrs);
  return retVal;
}

int
IPolysMemIn(buf, size, pIPolys)
     char *buf;
     int size;
     iPolys      *pIPolys;
{
  XDR             xdrs;
  int     retVal = 0;

  xdrmem_create(&xdrs, buf, size, XDR_DECODE);
```

```
  if(!xdr_iPolys(&xdrs, pIPolys)){
    retVal = -1;
  }
  xdr_destroy(&xdrs);
  return retVal;
}

void
inputIPolys(fp, pIPolys)
     FILE           *fp;
     iPolys      *pIPolys;
{
/*read from stream*/
  memset(pIPolys, 0, sizeof(iPolys));
}

void
outputIPolys(fp, pIPolys)
     FILE           *fp;
     iPolys      *pIPolys;
{
  int i;
  fprintf(fp, "/*N Ipolys*/\n");
  fprintf(fp, "%d\n");
  for(i=0; i<pIPolys->iPolys_len; i++){
    outputIPoly(fp, &pIPolys->iPolys_val[i]);
  }
}

void
freeIPolys(pIPolys)
     iPolys      *pIPolys;
{
  IPolysXDRFree(pIPolys);
}

iPolys      *
copyIPolys(pIPolys, destpIPolys)
     iPolys      *pIPolys;
     iPolys      *destpIPolys;
{
  char *buf;
  int bufSize = 65536;
  iPolys *newIPolys;

  buf = malloc(bufSize);

  while(IPolysMemOut(buf, bufSize, pIPolys)== -1){
    bufSize *=2;
    buf = realloc(buf, bufSize);
  }
  if(destpIPolys){
    newIPolys = destpIPolys;
```

```
  }
  else{
    newIPolys = (iPolys *)malloc(sizeof(iPolys));
    memset(newIPolys, 0, sizeof(iPolys));
  }
  IPolysMemIn(buf, bufSize, newIPolys);
  free(buf);

  return newIPolys;
}

void
IPolysXDRFree(pIPolys)
     iPolys      *pIPolys;
{
  xdr_free(xdr_iPolys, (char *) pIPolys);
  memset(pIPolys, 0, sizeof(iPolys));
}


#ifdef STANDALONE
main(argc, argv)
#else               /* STANDALONE */
     IPolyMain(argc, argv)
#endif              /* STANDALONE */
     int             argc;
     char          **argv;
{
  iPoly sIPoly;
  iPoly cpIPoly;
  iPolys sIPolys;
  iPolys cpIPolys;

  switch (argc) {
  case 1:       /* receive sId */
    IPolyIn(0 /* stdin */ , &sIPoly);
    outputIPoly(stdout, &sIPoly);
    cpIPoly = sIPoly;
    outputIPoly(stdout, &cpIPoly);

    break;
  case 2:       /* receive sId */
    inputIPoly(stdin, &sIPoly);
#ifdef DEBUG
    outputIPoly(stderr, &sIPoly);
#endif
    IPolyOut(1 /* stdout */ , &sIPoly);

    break;
  case 3:       /* receive sId */
    IPolysIn(0 /* stdin */ , &sIPolys);
    outputIPolys(stdout, &sIPolys);
    cpIPolys = sIPolys;
```

```
      outputIPolys(stdout, &cpIPolys);

      break;
    case 4:       /* receive sId */
      inputIPolys(stdin, &sIPolys);
#ifdef DEBUG
      outputIPolys(stderr, &sIPolys);
#endif
      IPolysOut(1 /* stdout */ , &sIPolys);

      break;
    }

}
```

```
/*****************************************************************************
    ***/
/*****************************************************************************
    ***/
/**
    **/
/** This SHASTRA software is not in the Public Domain. It is distributed on
    **/
/** a person to person basis, solely for educational use and permission is
    **/
/** NOT granted for its transfer to anyone or for its use in any commercial
    **/
/** product.  There is NO warranty on the available software and neither
    **/
/** Purdue University nor the Applied Algebra and Geometry group directed
    **/
/** by C.  Bajaj accept responsibility for the consequences of its use.
    **/
/**
    **/
/*****************************************************************************
    ***/
/*****************************************************************************
    ***/
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include <rpc/rpc.h>
#include <shastra/datacomm/iPoly.h>

bool_t
xdr_iPolyPoint(xdrs, objp)
    XDR *xdrs;
    iPolyPoint objp;
{
    if (!xdr_vector(xdrs, (char *)objp, 3, sizeof(double), xdr_double)) {
        return (FALSE);
    }
    return (TRUE);
}

bool_t
xdr_iPolySize(xdrs, objp)
    XDR *xdrs;
    iPolySize *objp;
{
    if (!xdr_double(xdrs, objp)) {
        return (FALSE);
    }
    return (TRUE);
}
```

```
bool_t
xdr_iPolyNormal(xdrs, objp)
    XDR *xdrs;
    iPolyNormal objp;
{
    if (!xdr_vector(xdrs, (char *)objp, 3, sizeof(float), xdr_float)) {
        return (FALSE);
    }
    return (TRUE);
}

bool_t
xdr_iPolyRGB(xdrs, objp)
    XDR *xdrs;
    iPolyRGB objp;
{
    if (!xdr_vector(xdrs, (char *)objp, 3, sizeof(float), xdr_float)) {
        return (FALSE);
    }
    return (TRUE);
}

bool_t
xdr_iPolyEdgeVerts(xdrs, objp)
    XDR *xdrs;
    iPolyEdgeVerts objp;
{
    if (!xdr_vector(xdrs, (char *)objp, 2, sizeof(int), xdr_int)) {
        return (FALSE);
    }
    return (TRUE);
}

bool_t
xdr_range(xdrs, objp)
    XDR *xdrs;
    range objp;
{
    if (!xdr_vector(xdrs, (char *)objp, 2, sizeof(int), xdr_int)) {
        return (FALSE);
    }
    return (TRUE);
}

bool_t
xdr_iRanges(xdrs, objp)
    XDR *xdrs;
    iRanges *objp;
{
    if (!xdr_array(xdrs, (char **)&objp->iRanges_val, (u_int *)&objp->
        iRanges_len, ~0, sizeof(range), xdr_range)) {
        return (FALSE);
```

```
        }
        return (TRUE);
    }

    bool_t
    xdr_iEdges(xdrs, objp)
        XDR *xdrs;
        iEdges *objp;
    {
        if (!xdr_array(xdrs, (char **)&objp->iEdges_val, (u_int *)&objp->
            iEdges_len, ~0, sizeof(int), xdr_int)) {
            return (FALSE);
        }
        return (TRUE);
    }

    bool_t
    xdr_iVerts(xdrs, objp)
        XDR *xdrs;
        iVerts *objp;
    {
        if (!xdr_array(xdrs, (char **)&objp->iVerts_val, (u_int *)&objp->
            iVerts_len, ~0, sizeof(int), xdr_int)) {
            return (FALSE);
        }
        return (TRUE);
    }

    bool_t
    xdr_iFaces(xdrs, objp)
        XDR *xdrs;
        iFaces *objp;
    {
        if (!xdr_array(xdrs, (char **)&objp->iFaces_val, (u_int *)&objp->
            iFaces_len, ~0, sizeof(int), xdr_int)) {
            return (FALSE);
        }
        return (TRUE);
    }

    bool_t
    xdr_colorCode(xdrs, objp)
        XDR *xdrs;
        colorCode *objp;
    {
        if (!xdr_enum(xdrs, (enum_t *)objp)) {
            return (FALSE);
        }
        return (TRUE);
    }

    bool_t
    xdr_iPolyColors(xdrs, objp)
```

```
    XDR *xdrs;
    iPolyColors *objp;
{
    if (!xdr_colorCode(xdrs, &objp->code)) {
        return (FALSE);
    }
    switch (objp->code) {
    case ColorIndex:
        if (!xdr_array(xdrs, (char **)&objp->iPolyColors_u.colors.
            colors_val, (u_int *)&objp->iPolyColors_u.colors.colors_len, ~0
            , sizeof(int), xdr_int)) {
            return (FALSE);
        }
        break;
    case ShadeIndex:
        if (!xdr_array(xdrs, (char **)&objp->iPolyColors_u.shades.
            shades_val, (u_int *)&objp->iPolyColors_u.shades.shades_len, ~0
            , sizeof(int), xdr_int)) {
            return (FALSE);
        }
        break;
    case ColorMapVal:
        if (!xdr_array(xdrs, (char **)&objp->iPolyColors_u.values.
            values_val, (u_int *)&objp->iPolyColors_u.values.values_len, ~0
            , sizeof(float), xdr_float)) {
            return (FALSE);
        }
        break;
    }
    return (TRUE);
}

bool_t
xdr_iPolyVerts(xdrs, objp)
    XDR *xdrs;
    iPolyVerts *objp;
{
    if (!xdr_array(xdrs, (char **)&objp->points.points_val, (u_int *)&objp-
        >points.points_len, ~0, sizeof(iPolyPoint), xdr_iPolyPoint)) {
        return (FALSE);
    }
    if (!xdr_array(xdrs, (char **)&objp->aFaces.aFaces_val, (u_int *)&objp-
        >aFaces.aFaces_len, ~0, sizeof(iFaces), xdr_iFaces)) {
        return (FALSE);
    }
    if (!xdr_array(xdrs, (char **)&objp->aEdges.aEdges_val, (u_int *)&objp-
        >aEdges.aEdges_len, ~0, sizeof(iEdges), xdr_iEdges)) {
        return (FALSE);
    }
    if (!xdr_array(xdrs, (char **)&objp->normals.normals_val, (u_int *)&
        objp->normals.normals_len, ~0, sizeof(iPolyNormal), xdr_iPolyNormal
        )) {
        return (FALSE);
```

```
        }
        if (!xdr_array(xdrs, (char **)&objp->sizes.sizes_val, (u_int *)&objp->
            sizes.sizes_len, ~0, sizeof(iPolySize), xdr_iPolySize)) {
            return (FALSE);
        }
        if (!xdr_iPolyColors(xdrs, &objp->colors)) {
            return (FALSE);
        }
        return (TRUE);
    }

    bool_t
    xdr_iPolyEdges(xdrs, objp)
        XDR *xdrs;
        iPolyEdges *objp;
    {
        if (!xdr_array(xdrs, (char **)&objp->verts.verts_val, (u_int *)&objp->
            verts.verts_len, ~0, sizeof(iPolyEdgeVerts), xdr_iPolyEdgeVerts)) {
            return (FALSE);
        }
        if (!xdr_array(xdrs, (char **)&objp->aFaces.aFaces_val, (u_int *)&objp-
            >aFaces.aFaces_len, ~0, sizeof(iFaces), xdr_iFaces)) {
            return (FALSE);
        }
        if (!xdr_array(xdrs, (char **)&objp->sizes.sizes_val, (u_int *)&objp->
            sizes.sizes_len, ~0, sizeof(iPolySize), xdr_iPolySize)) {
            return (FALSE);
        }
        if (!xdr_iPolyColors(xdrs, &objp->colors)) {
            return (FALSE);
        }
        return (TRUE);
    }

    bool_t
    xdr_iPolyFaces(xdrs, objp)
        XDR *xdrs;
        iPolyFaces *objp;
    {
        if (!xdr_array(xdrs, (char **)&objp->eFaces.eFaces_val, (u_int *)&objp-
            >eFaces.eFaces_len, ~0, sizeof(iEdges), xdr_iEdges)) {
            return (FALSE);
        }
        if (!xdr_array(xdrs, (char **)&objp->vFaces.vFaces_val, (u_int *)&objp-
            >vFaces.vFaces_len, ~0, sizeof(iVerts), xdr_iVerts)) {
            return (FALSE);
        }
        if (!xdr_array(xdrs, (char **)&objp->sizes.sizes_val, (u_int *)&objp->
            sizes.sizes_len, ~0, sizeof(iPolySize), xdr_iPolySize)) {
            return (FALSE);
        }
        if (!xdr_array(xdrs, (char **)&objp->normals.normals_val, (u_int *)&
            objp->normals.normals_len, ~0, sizeof(iPolyNormal), xdr_iPolyNormal
```

```
            )) {
            return (FALSE);
        }
        if (!xdr_iPolyColors(xdrs, &objp->colors)) {
            return (FALSE);
        }
        return (TRUE);
    }

    bool_t
    xdr_iPolyElmts(xdrs, objp)
        XDR *xdrs;
        iPolyElmts *objp;
    {
        if (!xdr_array(xdrs, (char **)&objp->fElmts.fElmts_val, (u_int *)&objp-
            >fElmts.fElmts_len, ~0, sizeof(iFaces), xdr_iFaces)) {
            return (FALSE);
        }
        if (!xdr_array(xdrs, (char **)&objp->sizes.sizes_val, (u_int *)&objp->
            sizes.sizes_len, ~0, sizeof(iPolySize), xdr_iPolySize)) {
            return (FALSE);
        }
        if (!xdr_iPolyColors(xdrs, &objp->colors)) {
            return (FALSE);
        }
        return (TRUE);
    }

    bool_t
    xdr_iPoly(xdrs, objp)
        XDR *xdrs;
        iPoly *objp;
    {
        if (!xdr_iPolyVerts(xdrs, &objp->verts)) {
            return (FALSE);
        }
        if (!xdr_iPolyEdges(xdrs, &objp->edges)) {
            return (FALSE);
        }
        if (!xdr_iPolyFaces(xdrs, &objp->faces)) {
            return (FALSE);
        }
        if (!xdr_iPolyElmts(xdrs, &objp->elmts)) {
            return (FALSE);
        }
        if (!xdr_array(xdrs, (char **)&objp->eGroups.eGroups_val, (u_int *)&
            objp->eGroups.eGroups_len, ~0, sizeof(iRanges), xdr_iRanges)) {
            return (FALSE);
        }
        if (!xdr_array(xdrs, (char **)&objp->fGroups.fGroups_val, (u_int *)&
            objp->fGroups.fGroups_len, ~0, sizeof(iRanges), xdr_iRanges)) {
            return (FALSE);
        }
```

```
    if (!xdr_array(xdrs, (char **)&objp->rgb.rgb_val, (u_int *)&objp->rgb.
        rgb_len, ~0, sizeof(iPolyRGB), xdr_iPolyRGB)) {
        return (FALSE);
    }
    return (TRUE);
}

bool_t
xdr_iPoly_P(xdrs, objp)
    XDR *xdrs;
    iPoly_P *objp;
{
    if (!xdr_pointer(xdrs, (char **)objp, sizeof(iPoly), xdr_iPoly)) {
        return (FALSE);
    }
    return (TRUE);
}

bool_t
xdr_iPolys(xdrs, objp)
    XDR *xdrs;
    iPolys *objp;
{
    if (!xdr_array(xdrs, (char **)&objp->iPolys_val, (u_int *)&objp->
        iPolys_len, ~0, sizeof(iPoly), xdr_iPoly)) {
        return (FALSE);
    }
    return (TRUE);
}

bool_t
xdr_iPolys_P(xdrs, objp)
    XDR *xdrs;
    iPolys_P *objp;
{
    if (!xdr_pointer(xdrs, (char **)objp, sizeof(iPolys), xdr_iPolys)) {
        return (FALSE);
    }
    return (TRUE);
}

bool_t
xdr_iPolyObj(xdrs, objp)
    XDR *xdrs;
    iPolyObj *objp;
{
    if (!xdr_vector(xdrs, (char *)objp->sbName, IPOLY_NMLEN, sizeof(char),
        xdr_char)) {
        return (FALSE);
    }
    if (!xdr_u_long(xdrs, &objp->lIdTag)) {
        return (FALSE);
    }
```

```
        if (!xdr_u_long(xdrs, &objp->lSIdTag)) {
            return (FALSE);
        }
        if (!xdr_u_long(xdrs, &objp->lPerms)) {
            return (FALSE);
        }
        if (!xdr_u_long(xdrs, &objp->lType)) {
            return (FALSE);
        }
        if (!xdr_u_long(xdrs, &objp->lMode)) {
            return (FALSE);
        }
        if (!xdr_pointer(xdrs, (char **)&objp->pIPoly, sizeof(iPoly), xdr_iPoly
            )) {
            return (FALSE);
        }
        return (TRUE);
    }

    bool_t
    xdr_iPolyObj_P(xdrs, objp)
        XDR *xdrs;
        iPolyObj_P *objp;
    {
        if (!xdr_pointer(xdrs, (char **)objp, sizeof(iPolyObj), xdr_iPolyObj))
            {
            return (FALSE);
        }
        return (TRUE);
    }
```

```
/*****************************************************************************
    ***/
/*****************************************************************************
    ***/
/**
    **/
/** This SHASTRA software is not in the Public Domain. It is distributed on
    **/
/** a person to person basis, solely for educational use and permission is
    **/
/** NOT granted for its transfer to anyone or for its use in any commercial
    **/
/** product.  There is NO warranty on the available software and neither
    **/
/** Purdue University nor the Applied Algebra and Geometry group directed
    **/
/** by C.  Bajaj accept responsibility for the consequences of its use.
    **/
/**
    **/
/*****************************************************************************
    ***/
/*****************************************************************************
    ***/
#include <stdio.h>

#include <shastra/datacomm/pictDataH.h>
#include <shastra/network/mplex.h>
#include <shastra/network/rpc.h>

#define STANDALONEnn

int
pictPieceOut(fd, pPictCData)
    int             fd;
    pictPiece   *pPictCData;
{
    XDR             xdrs;
    int             retVal = 0;

#ifdef STANDALONE
    {
        FILE            *fp;
        fp = stdout /* fdopen(fd,"w") */ ;
        xdrstdio_create(&xdrs, fp, XDR_ENCODE);
        if (!xdr_pictPiece(&xdrs, pPictCData)) {
            retVal = -1;
        }
    }
#else               /* STANDALONE */
    /*
     * xdrstdio_create(mplexXDRSEnc(fd), mplexOutStream(fd), XDR_ENCODE);
     */
```

```
    if (!xdr_pictPiece(mplexXDRSEnc(fd), pPictCData)) {
        retVal = -1;
    }
#endif              /* STANDALONE */
    return retVal;
}

int
pictPieceIn(fd, pPictCData)
    int             fd;
    pictPiece   *pPictCData;
{
    XDR             xdrs;
    int             retVal = 0;

    pictPieceXDRFree(pPictCData);
#ifdef STANDALONE
    {
        FILE           *fp;
        fp = stdin /* fdopen(fd,"r") */ ;
        xdrstdio_create(&xdrs, fp, XDR_DECODE);
        if (!xdr_pictPiece(&xdrs, pPictCData)) {
            retVal = -1;
        }
    }
#else               /* STANDALONE */
    /*
     * xdrstdio_create(mplexXDRSDec(fd), mplexInStream(fd), XDR_DECODE);
     */
    if (!xdr_pictPiece(mplexXDRSDec(fd), pPictCData)) {
        retVal = -1;
    }
#endif              /* STANDALONE */
    return retVal;
}

int
pictPieceMemOut(buf, size, pPictCData)
    char           *buf;
    int             size;
    pictPiece   *pPictCData;
{
    XDR             xdrs;
    int             retVal = 0;

    xdrmem_create(&xdrs, buf, size, XDR_ENCODE);
    if (!xdr_pictPiece(&xdrs, pPictCData)) {
        retVal = -1;
    }
    xdr_destroy(&xdrs);
    return retVal;
}
```

```
int
pictPieceMemIn(buf, size, pPictCData)
    char           *buf;
    int             size;
    pictPiece   *pPictCData;
{
    XDR             xdrs;
    int             retVal = 0;

    pictPieceXDRFree(pPictCData);
    xdrmem_create(&xdrs, buf, size, XDR_DECODE);
    if (!xdr_pictPiece(&xdrs, pPictCData)) {
        retVal = -1;
    }
    xdr_destroy(&xdrs);
    return retVal;
}

int
pictPiecesOut(fd, pPictCDatas)
    int             fd;
    pictPieces  *pPictCDatas;
{
    XDR             xdrs;
    int             retVal = 0;

#ifdef STANDALONE
    {
        FILE           *fp;
        fp = stdout /* fdopen(fd,"w") */ ;
        xdrstdio_create(&xdrs, fp, XDR_ENCODE);
        if (!xdr_pictPieces(&xdrs, pPictCDatas)) {
            retVal = -1;
        }
    }
#else               /* STANDALONE */
    /*
     * xdrstdio_create(mplexXDRSEnc(fd), mplexOutStream(fd), XDR_ENCODE);
     */
    if (!xdr_pictPieces(mplexXDRSEnc(fd), pPictCDatas)) {
        retVal = -1;
    }
#endif              /* STANDALONE */
    return retVal;
}

int
pictPiecesIn(fd, pPictCDatas)
    int             fd;
    pictPieces  *pPictCDatas;
{
    XDR             xdrs;
    int             retVal = 0;
```

```
    pictPiecesXDRFree(pPictCDatas);
#ifdef STANDALONE
    {
        FILE           *fp;
        fp = stdin /* fdopen(fd,"r") */ ;
        xdrstdio_create(&xdrs, fp, XDR_DECODE);
        if (!xdr_pictPieces(&xdrs, pPictCDatas)) {
            retVal = -1;
        }
    }
#else               /* STANDALONE */
    /*
     * xdrstdio_create(mplexXDRSDec(fd), mplexInStream(fd), XDR_DECODE);
     */
    if (!xdr_pictPieces(mplexXDRSDec(fd), pPictCDatas)) {
        retVal = -1;
    }
#endif              /* STANDALONE */
    return retVal;
}

int
pictPiecesMemOut(buf, size, pPictCDatas)
    char           *buf;
    int             size;
    pictPieces  *pPictCDatas;
{
    XDR             xdrs;
    int             retVal = 0;

    xdrmem_create(&xdrs, buf, size, XDR_ENCODE);
    if (!xdr_pictPieces(&xdrs, pPictCDatas)) {
        retVal = -1;
    }
    xdr_destroy(&xdrs);
    return retVal;
}

int
pictPiecesMemIn(buf, size, pPictCDatas)
    char           *buf;
    int             size;
    pictPieces  *pPictCDatas;
{
    XDR             xdrs;
    int             retVal = 0;

    pictPiecesXDRFree(pPictCDatas);
    xdrmem_create(&xdrs, buf, size, XDR_DECODE);
    if (!xdr_pictPieces(&xdrs, pPictCDatas)) {
        retVal = -1;
    }
```

```
    xdr_destroy(&xdrs);
    return retVal;
}

int
pictCollexnOut(fd, pPictCollexn)
    int             fd;
    pictCollexn  *pPictCollexn;
{
    XDR             xdrs;
    int             retVal = 0;

#ifdef STANDALONE
    {
        FILE           *fp;
        fp = stdout /* fdopen(fd,"w") */ ;
        xdrstdio_create(&xdrs, fp, XDR_ENCODE);
        if (!xdr_pictCollexn(&xdrs, pPictCollexn)) {
            retVal = -1;
        }
    }
#else               /* STANDALONE */
    /*
     * xdrstdio_create(mplexXDRSEnc(fd), mplexOutStream(fd), XDR_ENCODE);
     */
    if (!xdr_pictCollexn(mplexXDRSEnc(fd), pPictCollexn)) {
        retVal = -1;
    }
#endif              /* STANDALONE */
    return retVal;
}

int
pictCollexnIn(fd, pPictCollexn)
    int             fd;
    pictCollexn  *pPictCollexn;
{
    XDR             xdrs;
    int             retVal = 0;

    pictCollexnXDRFree(pPictCollexn);
#ifdef STANDALONE
    {
        FILE           *fp;
        fp = stdin /* fdopen(fd,"r") */ ;
        xdrstdio_create(&xdrs, fp, XDR_DECODE);
        if (!xdr_pictCollexn(&xdrs, pPictCollexn)) {
            retVal = -1;
        }
    }
#else               /* STANDALONE */
    /*
     * xdrstdio_create(mplexXDRSDec(fd), mplexInStream(fd), XDR_DECODE);
```

```
     */
    if (!xdr_pictCollexn(mplexXDRSDec(fd), pPictCollexn)) {
        retVal = -1;
    }
#endif              /* STANDALONE */
    return retVal;
}

int
pictCollexnMemOut(buf, size, pPictCollexn)
    char           *buf;
    int             size;
    pictCollexn  *pPictCollexn;
{
    XDR             xdrs;
    int             retVal = 0;

    xdrmem_create(&xdrs, buf, size, XDR_ENCODE);
    if (!xdr_pictCollexn(&xdrs, pPictCollexn)) {
        retVal = -1;
    }
    xdr_destroy(&xdrs);
    return retVal;
}

int
pictCollexnMemIn(buf, size, pPictCollexn)
    char           *buf;
    int             size;
    pictCollexn  *pPictCollexn;
{
    XDR             xdrs;
    int             retVal = 0;

    pictCollexnXDRFree(pPictCollexn);
    xdrmem_create(&xdrs, buf, size, XDR_DECODE);
    if (!xdr_pictCollexn(&xdrs, pPictCollexn)) {
        retVal = -1;
    }
    xdr_destroy(&xdrs);
    return retVal;
}

void
freePictPiece(pPictCData)
    pictPiece   *pPictCData;
{
    if (pPictCData == NULL) {
        return;
    }
    memset(pPictCData, 0, sizeof(pictPiece));
}
```

```
void
freePictPieces(pPictCDatas)
    pictPieces  *pPictCDatas;
{
    int             i;

    if (pPictCDatas == NULL) {
        return;
    }
    for (i = 0; i < pPictCDatas->pictPieces_len; i++) {
        freePictPiece(&pPictCDatas->pictPieces_val[i]);
    }
    free(pPictCDatas->pictPieces_val);
    memset(pPictCDatas, 0, sizeof(pictPieces));
}

pictPiece   *
copyPictPiece(pPictCData, destpPictCData)
    pictPiece   *pPictCData;
    pictPiece   *destpPictCData;
{
    pictPiece   *newpPictCData;
    int             i;

    if (pPictCData == NULL) {
        return NULL;
    }
    if (destpPictCData == NULL) {
        newpPictCData = (pictPiece *) malloc(sizeof(pictPiece));
    } else {
        newpPictCData = destpPictCData;
    }

    memcpy(newpPictCData, pPictCData, sizeof(pictPiece));
    return newpPictCData;
}

pictPieces  *
copyPictPieces(pPictCDatas, destpPictCDatas)
    pictPieces  *pPictCDatas;
    pictPieces  *destpPictCDatas;
{
    int             i;
    pictPieces  *newpPictCDatas;
    char            buf[65536];

    if (pPictCDatas == NULL) {
        return NULL;
    }
    if (destpPictCDatas == NULL) {
        newpPictCDatas = (pictPieces *) malloc(sizeof(pictPieces));
        memset(newpPictCDatas, 0, sizeof(pictPieces));
```

```
    } else {
        newpPictCDatas = destpPictCDatas;
    }
    pictPiecesMemOut(buf, 65536, pPictCDatas);
    pictPiecesMemIn(buf, 65536, newpPictCDatas);
    return newpPictCDatas;
}


void
inputPictPiece(fp, pPictCData)
    FILE            *fp;
    pictPiece   *pPictCData;
{
    memset(pPictCData, 0, sizeof(pictPiece));
}

void
outputPictPiece(fp, pPictCData)
    FILE            *fp;
    pictPiece   *pPictCData;
{
    fprintf(stderr, "outputPictPiece() not complete\n");
}


void
inputPictPieces(fp, pPictCDatas)
    FILE            *fp;
    pictPieces  *pPictCDatas;
{
    int             i;

    fscanf(fp, "%d", &pPictCDatas->pictPieces_len);
    pPictCDatas->pictPieces_val = (pictPiece *)
        malloc(pPictCDatas->pictPieces_len * sizeof(pictPiece));
    for (i = 0; i < pPictCDatas->pictPieces_len; i++) {
        inputPictPiece(fp, &pPictCDatas->pictPieces_val[i]);
    }
}

void
outputPictPieces(fp, pPictCDatas)
    FILE            *fp;
    pictPieces  *pPictCDatas;
{
    int             i;

    fprintf(fp, "%d\n", pPictCDatas->pictPieces_len);
    for (i = 0; i < pPictCDatas->pictPieces_len; i++) {
        outputPictPiece(fp, &pPictCDatas->pictPieces_val[i]);
    }
```

```
}

void
pictPieceXDRFree(pPictCData)
    pictPiece   *pPictCData;
{
    xdr_free(xdr_pictPiece, (char *) pPictCData);
    memset(pPictCData, 0, sizeof(pictPiece));
}

void
pictPiecesXDRFree(pPictCDatas)
    pictPieces  *pPictCDatas;
{
    xdr_free(xdr_pictPieces, (char *) pPictCDatas);
    memset(pPictCDatas, 0, sizeof(pictPieces));
}

void
pictCollexnXDRFree(pPictCollexn)
    pictCollexn  *pPictCollexn;
{
    xdr_free(xdr_pictCollexn, (char *) pPictCollexn);
    memset(pPictCollexn, 0, sizeof(pictCollexn));
}

#ifdef STANDALONE
main(argc, argv)
#else               /* STANDALONE */
pictPieceMain(argc, argv)
#endif              /* STANDALONE */
    int             argc;
    char          **argv;
{
    static pictPiece pictCData;
    static pictPieces pictCDatas;
    pictPieces  *cpPictCDatas;
    pictPiece   *cpPictCData;

    switch (argc) {
    case 1:     /* receive pictPiece */
        pictPieceIn(0 /* stdin */ , &pictCData);
        outputPictPiece(stdout, &pictCData);
        cpPictCData = copyPictPiece(&pictCData, NULL);
        outputPictPiece(stdout, cpPictCData);
        freePictPiece(cpPictCData);

        break;
    case 2:     /* receive pictPiece */
       inputPictPiece(stdin, &pictCData);
#ifdef DEBUG
        outputPictPiece(stderr, &pictCData);
#endif
```

```
            pictPieceOut(1 /* stdout */ , &pictCData);

            break;
        case 3:     /* receive pictPieces */
            pictPiecesIn(0 /* stdin */ , &pictCDatas);
            outputPictPieces(stdout, &pictCDatas);
            cpPictCDatas = copyPictPieces(&pictCDatas, NULL);
            outputPictPieces(stdout, cpPictCDatas);
            freePictPieces(cpPictCDatas);

            break;
        case 4:     /* receive pictPieces */
            inputPictPieces(stdin, &pictCDatas);
    #ifdef DEBUG
            outputPictPieces(stderr, &pictCDatas);
    #endif
            pictPiecesOut(1 /* stdout */ , &pictCDatas);

            break;
        }

    }
```

```
/*****************************************************************************
    ***/
/*****************************************************************************
    ***/
/**
    **/
/** This SHASTRA software is not in the Public Domain. It is distributed on
    **/
/** a person to person basis, solely for educational use and permission is
    **/
/** NOT granted for its transfer to anyone or for its use in any commercial
    **/
/** product.  There is NO warranty on the available software and neither
    **/
/** Purdue University nor the Applied Algebra and Geometry group directed
    **/
/** by C.  Bajaj accept responsibility for the consequences of its use.
    **/
/**
    **/
/*****************************************************************************
    ***/
/*****************************************************************************
    ***/
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include <rpc/rpc.h>
#include <shastra/datacomm/xsCntlData.h>
#include <shastra/datacomm/pictData.h>

bool_t
xdr_lineMode(xdrs, objp)
    XDR *xdrs;
    lineMode *objp;
{
    if (!xdr_enum(xdrs, (enum_t *)objp)) {
        return (FALSE);
    }
    return (TRUE);
}

bool_t
xdr_mLineMode(xdrs, objp)
    XDR *xdrs;
    mLineMode *objp;
{
    if (!xdr_enum(xdrs, (enum_t *)objp)) {
        return (FALSE);
    }
    return (TRUE);
```

```
}

bool_t
xdr_objectMode(xdrs, objp)
    XDR *xdrs;
    objectMode *objp;
{
    if (!xdr_enum(xdrs, (enum_t *)objp)) {
        return (FALSE);
    }
    return (TRUE);
}

bool_t
xdr_editModes(xdrs, objp)
    XDR *xdrs;
    editModes *objp;
{
    if (!xdr_enum(xdrs, (enum_t *)objp)) {
        return (FALSE);
    }
    return (TRUE);
}

bool_t
xdr_pictMode(xdrs, objp)
    XDR *xdrs;
    pictMode *objp;
{
    if (!xdr_enum(xdrs, (enum_t *)objp)) {
        return (FALSE);
    }
    return (TRUE);
}

bool_t
xdr_pntData(xdrs, objp)
    XDR *xdrs;
    pntData objp;
{
    if (!xdr_vector(xdrs, (char *)objp, 3, sizeof(double), xdr_double)) {
        return (FALSE);
    }
    return (TRUE);
}

bool_t
xdr_unitData(xdrs, objp)
    XDR *xdrs;
    unitData *objp;
{
    if (!xdr_vector(xdrs, (char *)objp->start, 3, sizeof(double),
        xdr_double)) {
```

```
            return (FALSE);
        }
        if (!xdr_vector(xdrs, (char *)objp->end, 3, sizeof(double), xdr_double)
            ) {
            return (FALSE);
        }
        return (TRUE);
    }

    bool_t
    xdr_rndData(xdrs, objp)
        XDR *xdrs;
        rndData *objp;
    {
        if (!xdr_vector(xdrs, (char *)objp->start, 3, sizeof(double),
            xdr_double)) {
            return (FALSE);
        }
        if (!xdr_vector(xdrs, (char *)objp->end, 3, sizeof(double), xdr_double)
            ) {
            return (FALSE);
        }
        if (!xdr_double(xdrs, &objp->factor)) {
            return (FALSE);
        }
        return (TRUE);
    }

    bool_t
    xdr_pllgmData(xdrs, objp)
        XDR *xdrs;
        pllgmData *objp;
    {
        if (!xdr_vector(xdrs, (char *)objp->start, 3, sizeof(double),
            xdr_double)) {
            return (FALSE);
        }
        if (!xdr_vector(xdrs, (char *)objp->end, 3, sizeof(double), xdr_double)
            ) {
            return (FALSE);
        }
        if (!xdr_double(xdrs, &objp->factor)) {
            return (FALSE);
        }
        return (TRUE);
    }

    bool_t
    xdr_modeLineData(xdrs, objp)
        XDR *xdrs;
        modeLineData *objp;
    {
        if (!xdr_vector(xdrs, (char *)objp->start, 3, sizeof(double),
```

```
            xdr_double)) {
            return (FALSE);
        }
        if (!xdr_vector(xdrs, (char *)objp->end, 3, sizeof(double), xdr_double)
            ) {
            return (FALSE);
        }
        if (!xdr_lineMode(xdrs, &objp->mode)) {
            return (FALSE);
        }
        return (TRUE);
    }

    bool_t
    xdr_angLineData(xdrs, objp)
        XDR *xdrs;
        angLineData *objp;
    {
        if (!xdr_vector(xdrs, (char *)objp->start, 3, sizeof(double),
            xdr_double)) {
            return (FALSE);
        }
        if (!xdr_vector(xdrs, (char *)objp->end, 3, sizeof(double), xdr_double)
            ) {
            return (FALSE);
        }
        if (!xdr_int(xdrs, &objp->angle)) {
            return (FALSE);
        }
        return (TRUE);
    }

    bool_t
    xdr_arcData(xdrs, objp)
        XDR *xdrs;
        arcData *objp;
    {
        if (!xdr_vector(xdrs, (char *)objp->start, 3, sizeof(double),
            xdr_double)) {
            return (FALSE);
        }
        if (!xdr_vector(xdrs, (char *)objp->end, 3, sizeof(double), xdr_double)
            ) {
            return (FALSE);
        }
        if (!xdr_int(xdrs, &objp->angStart)) {
            return (FALSE);
        }
        if (!xdr_int(xdrs, &objp->angSpan)) {
            return (FALSE);
        }
        return (TRUE);
    }
```

```
bool_t
xdr_nGonData(xdrs, objp)
    XDR *xdrs;
    nGonData *objp;
{
    if (!xdr_vector(xdrs, (char *)objp->start, 3, sizeof(double),
        xdr_double)) {
        return (FALSE);
    }
    if (!xdr_vector(xdrs, (char *)objp->end, 3, sizeof(double), xdr_double)
        ) {
        return (FALSE);
    }
    if (!xdr_int(xdrs, &objp->n)) {
        return (FALSE);
    }
    return (TRUE);
}

bool_t
xdr_pictElmt(xdrs, objp)
    XDR *xdrs;
    pictElmt *objp;
{
    if (!xdr_pictMode(xdrs, &objp->picType)) {
        return (FALSE);
    }
    switch (objp->picType) {
    case pmNULL:
        break;
    case pmDUMMY_UNIT:
        if (!xdr_unitData(xdrs, &objp->pictElmt_u.unit)) {
            return (FALSE);
        }
        break;
    case pmDUMMY_HYBR:
        if (!xdr_nGonData(xdrs, &objp->pictElmt_u.hybrid)) {
            return (FALSE);
        }
        break;
    case pmDUMMY_AGGR:
        if (!xdr_array(xdrs, (char **)&objp->pictElmt_u.aggreg.aggreg_val,
            (u_int *)&objp->pictElmt_u.aggreg.aggreg_len, ~0, sizeof
            (pntData), xdr_pntData)) {
            return (FALSE);
        }
        break;
    case pmDUMMY_COMP:
        if (!xdr_array(xdrs, (char **)&objp->pictElmt_u.compound.
            compound_val, (u_int *)&objp->pictElmt_u.compound.compound_len,
            ~0, sizeof(unitData), xdr_unitData)) {
            return (FALSE);
```

```
        }
        break;
    case pmDUMMY_LAM:
        if (!xdr_array(xdrs, (char **)&objp->pictElmt_u.dumlam.dumlam_val,
            (u_int *)&objp->pictElmt_u.dumlam.dumlam_len, ~0, sizeof
            (pntData), xdr_pntData)) {
            return (FALSE);
        }
        break;
    case pmPOINT:
        if (!xdr_pntData(xdrs, objp->pictElmt_u.point)) {
            return (FALSE);
        }
        break;
    case pmLINE:
        if (!xdr_unitData(xdrs, &objp->pictElmt_u.line)) {
            return (FALSE);
        }
        break;
    case pmH_LINE:
        if (!xdr_unitData(xdrs, &objp->pictElmt_u.hline)) {
            return (FALSE);
        }
        break;
    case pmV_LINE:
        if (!xdr_unitData(xdrs, &objp->pictElmt_u.vline)) {
            return (FALSE);
        }
        break;
    case pmA_LINE:
        if (!xdr_angLineData(xdrs, &objp->pictElmt_u.aline)) {
            return (FALSE);
        }
        break;
    case pmP_LINE:
        if (!xdr_angLineData(xdrs, &objp->pictElmt_u.pline)) {
            return (FALSE);
        }
        break;
    case pmMULTI_LINE:
        if (!xdr_array(xdrs, (char **)&objp->pictElmt_u.lines.lines_val,
            (u_int *)&objp->pictElmt_u.lines.lines_len, ~0, sizeof(pntData)
            , xdr_pntData)) {
            return (FALSE);
        }
        break;
    case pmMULTI_SEG:
        if (!xdr_array(xdrs, (char **)&objp->pictElmt_u.segs.segs_val,
            (u_int *)&objp->pictElmt_u.segs.segs_len, ~0, sizeof(unitData),
            xdr_unitData)) {
            return (FALSE);
        }
        break;
```

```
case pmPOLYGON:
    if (!xdr_array(xdrs, (char **)&objp->pictElmt_u.polygon.polygon_val
        , (u_int *)&objp->pictElmt_u.polygon.polygon_len, ~0, sizeof
        (pntData), xdr_pntData)) {
        return (FALSE);
    }
    break;
case pmRECTANGLE:
    if (!xdr_unitData(xdrs, &objp->pictElmt_u.rect)) {
        return (FALSE);
    }
    break;
case pmREGNGON:
    if (!xdr_nGonData(xdrs, &objp->pictElmt_u.ngon)) {
        return (FALSE);
    }
    break;
case pmSQUARE:
    if (!xdr_unitData(xdrs, &objp->pictElmt_u.square)) {
        return (FALSE);
    }
    break;
case pmELLIPSE:
    if (!xdr_unitData(xdrs, &objp->pictElmt_u.ellipse)) {
        return (FALSE);
    }
    break;
case pmCIRCLE:
    if (!xdr_unitData(xdrs, &objp->pictElmt_u.circle)) {
        return (FALSE);
    }
    break;
case pmDIAMOND:
    if (!xdr_unitData(xdrs, &objp->pictElmt_u.diamond)) {
        return (FALSE);
    }
    break;
case pmPLLOGRAM:
    if (!xdr_pllgmData(xdrs, &objp->pictElmt_u.pllgm)) {
        return (FALSE);
    }
    break;
case pmRNDSQUARE:
    if (!xdr_rndData(xdrs, &objp->pictElmt_u.rndsq)) {
        return (FALSE);
    }
    break;
case pmRNDRECT:
    if (!xdr_rndData(xdrs, &objp->pictElmt_u.rndrect)) {
        return (FALSE);
    }
    break;
case pmPRNDSQUARE:
```

```
        if (!xdr_rndData(xdrs, &objp->pictElmt_u.prndsq)) {
            return (FALSE);
        }
        break;
    case pmPRNDRECT:
        if (!xdr_rndData(xdrs, &objp->pictElmt_u.prndrect)) {
            return (FALSE);
        }
        break;
    case pmARNDSQUARE:
        if (!xdr_rndData(xdrs, &objp->pictElmt_u.arndsq)) {
            return (FALSE);
        }
        break;
    case pmARNDRECT:
        if (!xdr_rndData(xdrs, &objp->pictElmt_u.arndrect)) {
            return (FALSE);
        }
        break;
    case pmBERN_MLINE:
        if (!xdr_array(xdrs, (char **)&objp->pictElmt_u.bernline.
            bernline_val, (u_int *)&objp->pictElmt_u.bernline.bernline_len,
            ~0, sizeof(pntData), xdr_pntData)) {
            return (FALSE);
        }
        break;
    case pmBERN_LAMINA:
        if (!xdr_array(xdrs, (char **)&objp->pictElmt_u.bernlam.bernlam_val
            , (u_int *)&objp->pictElmt_u.bernlam.bernlam_len, ~0, sizeof
            (pntData), xdr_pntData)) {
            return (FALSE);
        }
        break;
    case pmSPLINE_MLINE:
        if (!xdr_array(xdrs, (char **)&objp->pictElmt_u.splineline.
            splineline_val, (u_int *)&objp->pictElmt_u.splineline.
            splineline_len, ~0, sizeof(pntData), xdr_pntData)) {
            return (FALSE);
        }
        break;
    case pmSPLINE_LAMINA:
        if (!xdr_array(xdrs, (char **)&objp->pictElmt_u.splinelam.
            splinelam_val, (u_int *)&objp->pictElmt_u.splinelam.
            splinelam_len, ~0, sizeof(pntData), xdr_pntData)) {
            return (FALSE);
        }
        break;
    case pmFREE_MLINE:
        if (!xdr_array(xdrs, (char **)&objp->pictElmt_u.freeline.
            freeline_val, (u_int *)&objp->pictElmt_u.freeline.freeline_len,
            ~0, sizeof(pntData), xdr_pntData)) {
            return (FALSE);
        }
```

```
    break;
case pmFREE_LAMINA:
    if (!xdr_array(xdrs, (char **)&objp->pictElmt_u.freelam.freelam_val
        , (u_int *)&objp->pictElmt_u.freelam.freelam_len, ~0, sizeof
        (pntData), xdr_pntData)) {
        return (FALSE);
    }
    break;
case pmCVRECTANGLE:
    if (!xdr_unitData(xdrs, &objp->pictElmt_u.cvrect)) {
        return (FALSE);
    }
    break;
case pmCVREGNGON:
    if (!xdr_nGonData(xdrs, &objp->pictElmt_u.cvngon)) {
        return (FALSE);
    }
    break;
case pmCVSQUARE:
    if (!xdr_unitData(xdrs, &objp->pictElmt_u.cvsq)) {
        return (FALSE);
    }
    break;
case pmCVELLIPSE:
    if (!xdr_unitData(xdrs, &objp->pictElmt_u.cvell)) {
        return (FALSE);
    }
    break;
case pmCVCIRCLE:
    if (!xdr_unitData(xdrs, &objp->pictElmt_u.cvcircle)) {
        return (FALSE);
    }
    break;
case pmCVDIAMOND:
    if (!xdr_unitData(xdrs, &objp->pictElmt_u.cvdiamond)) {
        return (FALSE);
    }
    break;
case pmCVPLLOGRAM:
    if (!xdr_pllgmData(xdrs, &objp->pictElmt_u.cvpllgm)) {
        return (FALSE);
    }
    break;
case pmCVRNDSQUARE:
    if (!xdr_rndData(xdrs, &objp->pictElmt_u.cvrndsq)) {
        return (FALSE);
    }
    break;
case pmCVRNDRECT:
    if (!xdr_rndData(xdrs, &objp->pictElmt_u.cvrndrect)) {
        return (FALSE);
    }
    break;
```

```
        case pmCVPRNDSQUARE:
            if (!xdr_rndData(xdrs, &objp->pictElmt_u.cvprndsq)) {
                return (FALSE);
            }
            break;
        case pmCVPRNDRECT:
            if (!xdr_rndData(xdrs, &objp->pictElmt_u.cvprndrect)) {
                return (FALSE);
            }
            break;
        case pmCVARNDSQUARE:
            if (!xdr_rndData(xdrs, &objp->pictElmt_u.cvarndsq)) {
                return (FALSE);
            }
            break;
        case pmCVARNDRECT:
            if (!xdr_rndData(xdrs, &objp->pictElmt_u.cvarndrect)) {
                return (FALSE);
            }
            break;
        case pmCRREGNGON:
            if (!xdr_nGonData(xdrs, &objp->pictElmt_u.crngon)) {
                return (FALSE);
            }
            break;
        case pmCRSQUARE:
            if (!xdr_unitData(xdrs, &objp->pictElmt_u.crsq)) {
                return (FALSE);
            }
            break;
        case pmCRRNDSQUARE:
            if (!xdr_rndData(xdrs, &objp->pictElmt_u.crrndsq)) {
                return (FALSE);
            }
            break;
        case pmCRPRNDSQUARE:
            if (!xdr_rndData(xdrs, &objp->pictElmt_u.crprndsq)) {
                return (FALSE);
            }
            break;
        case pmCRARNDSQUARE:
            if (!xdr_rndData(xdrs, &objp->pictElmt_u.crarndsq)) {
                return (FALSE);
            }
            break;
        case pmCRCIRCLE:
            if (!xdr_unitData(xdrs, &objp->pictElmt_u.crcircle)) {
                return (FALSE);
            }
            break;
        case pmMNH_MLINE:
            if (!xdr_array(xdrs, (char **)&objp->pictElmt_u.mnhmline.
                mnhmline_val, (u_int *)&objp->pictElmt_u.mnhmline.mnhmline_len,
```

```
            ~0, sizeof(pntData), xdr_pntData)) {
            return (FALSE);
        }
        break;
    case pmMNH_POLYGON:
        if (!xdr_array(xdrs, (char **)&objp->pictElmt_u.mnhpoly.mnhpoly_val
            , (u_int *)&objp->pictElmt_u.mnhpoly.mnhpoly_len, ~0, sizeof
            (pntData), xdr_pntData)) {
            return (FALSE);
        }
        break;
    case pmMTN_MLINE:
        if (!xdr_array(xdrs, (char **)&objp->pictElmt_u.mtnmline.
            mtnmline_val, (u_int *)&objp->pictElmt_u.mtnmline.mtnmline_len,
            ~0, sizeof(pntData), xdr_pntData)) {
            return (FALSE);
        }
        break;
    case pmMTN_POLYGON:
        if (!xdr_array(xdrs, (char **)&objp->pictElmt_u.mtnpoly.mtnpoly_val
            , (u_int *)&objp->pictElmt_u.mtnpoly.mtnpoly_len, ~0, sizeof
            (pntData), xdr_pntData)) {
            return (FALSE);
        }
        break;
    case pmMTNMNH_MLINE:
        if (!xdr_array(xdrs, (char **)&objp->pictElmt_u.mtnmnhmline.
            mtnmnhmline_val, (u_int *)&objp->pictElmt_u.mtnmnhmline.
            mtnmnhmline_len, ~0, sizeof(pntData), xdr_pntData)) {
            return (FALSE);
        }
        break;
    case pmMTNMNH_POLYGON:
        if (!xdr_array(xdrs, (char **)&objp->pictElmt_u.mtnmnhpoly.
            mtnmnhpoly_val, (u_int *)&objp->pictElmt_u.mtnmnhpoly.
            mtnmnhpoly_len, ~0, sizeof(pntData), xdr_pntData)) {
            return (FALSE);
        }
        break;
    case pmTEXT:
        if (!xdr_unitData(xdrs, &objp->pictElmt_u.text)) {
            return (FALSE);
        }
        break;
    case pmTEXTPOLYGON:
        if (!xdr_unitData(xdrs, &objp->pictElmt_u.textpoly)) {
            return (FALSE);
        }
        break;
    case pmFTEXTPOLYGON:
        if (!xdr_unitData(xdrs, &objp->pictElmt_u.ftextpoly)) {
            return (FALSE);
        }
```

```
            break;
        case pmGRAPH:
            if (!xdr_array(xdrs, (char **)&objp->pictElmt_u.graph.graph_val,
                (u_int *)&objp->pictElmt_u.graph.graph_len, ~0, sizeof(pntData)
                , xdr_pntData)) {
                return (FALSE);
            }
            break;
        case pmFLOWCHART:
            if (!xdr_array(xdrs, (char **)&objp->pictElmt_u.flowchart.
                flowchart_val, (u_int *)&objp->pictElmt_u.flowchart.
                flowchart_len, ~0, sizeof(pntData), xdr_pntData)) {
                return (FALSE);
            }
            break;
        case pmARC:
            if (!xdr_arcData(xdrs, &objp->pictElmt_u.arc)) {
                return (FALSE);
            }
            break;
        case pmCVARC:
            if (!xdr_arcData(xdrs, &objp->pictElmt_u.cvarc)) {
                return (FALSE);
            }
            break;
        case pmCARC:
            if (!xdr_arcData(xdrs, &objp->pictElmt_u.carc)) {
                return (FALSE);
            }
            break;
        case pmCVCARC:
            if (!xdr_arcData(xdrs, &objp->pictElmt_u.cvcarc)) {
                return (FALSE);
            }
            break;
        case pmCRCARC:
            if (!xdr_arcData(xdrs, &objp->pictElmt_u.crcarc)) {
                return (FALSE);
            }
            break;
        case pmFRENCH:
            if (!xdr_unitData(xdrs, &objp->pictElmt_u.french)) {
                return (FALSE);
            }
            break;
        }
        return (TRUE);
    }

    bool_t
    xdr_pictPiece(xdrs, objp)
        XDR *xdrs;
        pictPiece *objp;
```

```
{
    if (!xdr_objId(xdrs, &objp->objectId)) {
        return (FALSE);
    }
    if (!xdr_pictElmt(xdrs, &objp->pict)) {
        return (FALSE);
    }
    return (TRUE);
}

bool_t
xdr_pictPiece_P(xdrs, objp)
    XDR *xdrs;
    pictPiece_P *objp;
{
    if (!xdr_pointer(xdrs, (char **)objp, sizeof(pictPiece), xdr_pictPiece)
        ) {
        return (FALSE);
    }
    return (TRUE);
}

bool_t
xdr_pictPieces(xdrs, objp)
    XDR *xdrs;
    pictPieces *objp;
{
    if (!xdr_array(xdrs, (char **)&objp->pictPieces_val, (u_int *)&objp->
        pictPieces_len, ~0, sizeof(pictPiece), xdr_pictPiece)) {
        return (FALSE);
    }
    return (TRUE);
}

bool_t
xdr_pictPieces_P(xdrs, objp)
    XDR *xdrs;
    pictPieces_P *objp;
{
    if (!xdr_pointer(xdrs, (char **)objp, sizeof(pictPieces),
        xdr_pictPieces)) {
        return (FALSE);
    }
    return (TRUE);
}

bool_t
xdr_pictCollexn(xdrs, objp)
    XDR *xdrs;
    pictCollexn *objp;
{
    if (!xdr_vector(xdrs, (char *)objp->sbName, 32, sizeof(char), xdr_char)
        ) {
```

```
            return (FALSE);
        }
        if (!xdr_u_long(xdrs, &objp->lIdTag)) {
            return (FALSE);
        }
        if (!xdr_u_long(xdrs, &objp->lSIdTag)) {
            return (FALSE);
        }
        if (!xdr_u_long(xdrs, &objp->lPerms)) {
            return (FALSE);
        }
        if (!xdr_long(xdrs, &objp->lType)) {
            return (FALSE);
        }
        if (!xdr_long(xdrs, &objp->lPointer)) {
            return (FALSE);
        }
        if (!xdr_pointer(xdrs, (char **)&objp->pPPieces, sizeof(pictPieces),
            xdr_pictPieces)) {
            return (FALSE);
        }
        return (TRUE);
    }
```